Security Audit Report

Apex Q1 2025: Reactor and Skyline critical path

Last revised 01.05.2025

Authors: Aleksandar Ignjatijevic, Simon Noetzlin, Carlos Rodriguez

Contents

Audit overview

The Project

In March and April 2025, Apex Foundation engaged Informal Systems to conduct a security audit of several scopes:

Reactor bridge

- `batcher` in apex-bridge
- `indexer` in cardano-infrastructure
- The bridge smart contracts in apex-bridge-smartcontracts
- The following e2e tests in blade-apex-bridge
  - apex_bridge_test.go
  - nexus_bridge_test.go
  - testnet_bridge_test.go

Skyline bridge

- `batcher` in apex-bridge
- The bridge smart contracts in apex-bridge-smartcontracts
- The following e2e tests in blade-apex-bridge
  - apex_bridge_test.go
  - nexus_bridge_test.go
  - testnet_bridge_test.go

Scope of this report

The audit focused on the components of the Reactor and Skyline bridges that the development team deemed as most critical: the Cardano block indexer, the off-chain batcher component and the bridge smart contracts. We reviewed correctness and security properties of the mentioned parts.

Audit plan

The audit was conducted between March 13th, 2025 and April 25th, 2025 by the following personnel:

- Aleksandar Ignatijevic
- Simon Noetzlin
- Carlos Rodriguez

Conclusions

The **Apex Reactor** and **Skyline** bridges connect the Apex Fusion ecosystem—among its constituent chains and with Cardano, respectively. These bridges form a complex system that operates as blockchains with a validator set running several off-chain components. In this audit, we focused on the parts of the system the development team identified as the highest priority: the Cardano indexer, batcher, and bridge smart contracts.

The system is highly asynchronous and distributed; its correct functioning depends on all moving parts working well together. We dedicated significant time to reasoning through scenarios that could raise safety or liveness issues. Most of these were unlikely given the design choices and trust assumptions (e.g., trusted off-chain oracle and batcher components, honest behavior of contract administrators, and a designated validator set). However, we did identify three critical issues: one that could have halted cross-chain transactions, and two that could have reduced the

system's Byzantine fault tolerance. Upon disclosure, the development team acknowledged the issues and promptly implemented fixes.

Overall, the project demonstrates a well-thought-out design and a high-quality implementation. While we reported several code improvements, we also recommend enhancing code documentation, as many functions lacked adequate comments. The cooperation and expertise of the development team significantly contributed to the success of this audit, and we believe the project is on a solid foundation going forward.

Audit Dashboard

Target Summary

- **Type:** protocol and implementation
- **Platform:** Go, Solidity
- **Artifacts:**
  - Reactor bridge:
    - The `batcher` in apex-bridge at commit hash `0dd6cc0`.
    - The `indexer` in cardano-infrastructure at commit hash `b150d63`.
    - The bridge smart contracts in apex-bridge-smartcontracts at commit hash `422fc3f`.
    - e2e tests in blade-apex-bridge at commit hash `66247f3`:
      - apex_bridge_test.go
      - nexus_bridge_test.go
      - testnet_bridge_test.go
  - Skyline bridge:
    - The `batcher` in apex-bridge at commit hash `e527566`.
    - No changes for the `indexer` with respect to the Reactor implementation at commit hash `ed906ee`.
    - The bridge smart contracts in apex-bridge-smartcontracts at commit hash `2392a60`.
    - e2e tests in blade-apex-bridge at commit hash `a3a5962`:
      - apex_bridge_test.go
      - nexus_bridge_test.go
      - testnet_bridge_test.go

Engagement Summary

- **Dates**: 24.03.2025 - 25.04.2025.
- **Method**: code review

Severity Summary

Finding Severity	Number
Critical	3
High	2
Medium	7
Low	4
Informational	14
Total	30

System Overview

The Apex Fusion ecosystem consists of a three-chain architecture designed to optimize scalability, security, and decentralization. Each of these interconnected blockchain networks serve a unique purpose.

At its core is the **Prime network**, the foundational layer that provides security and decentralization. Based on Cardano, it uses the Ouroboros Proof of Stake (PoS) consensus protocol and an e-UTXO accounting system for energy-efficient, robust security through decentralized liquid staking. The Prime chain is also the exclusive issuer of the APEX token, used for transaction fees, staking rewards, and governance, with token holders voting on proposals.

Supporting the Prime chain are two Layer 2 solutions:

1. **Vector network**: a secondary layer that enhances scalability and performance, designed for high-throughput applications and services. The Vector chain uses a UTXO model to deliver high throughput and low latency.
2. **Nexus network**: a secondary layer focused on speed and cost-efficiency. It handles smart contract execution and complex transactions. The Nexus chain uses an EVM-based framework, making it compatible with many decentralized applications and ensuring efficient DeFi interactions.

These networks are interconnected through the **Reactor** bridge, enabling seamless interoperability across the Apex Fusion ecosystem. This architecture addresses the blockchain trilemma by providing dedicated blockchains optimized for specific uses, offering users greater flexibility.

Reactor bridge

The Apex Reactor bridge connects the three chains in the Apex Fusion ecosystem: the Layer 1 Prime chain and two Layer 2 chains (Vector and Nexus). The bridge operates as a blockchain with a validator set. During token bridging, the system locks APEX tokens on the source blockchain in a multisig address controlled by bridge validators and releases an equivalent amount on the destination blockchain. Each validator runs two (trusted) off-chain components:

- The **oracle** monitors bridging activities, including bridging requests and batch executions. It detects bridging requests by observing transactions that deposit funds to the bridge multisig address, which controls the bridging funds.
- The **batcher** monitors the bridge blockchain to determine when to create a new batch. When specific conditions are met, the batcher creates a batch instance—a transaction intended for the destination blockchain.

To submit the batch to the destination chain, a trustless component called the **relayer** operates as a single standalone instance. The relayer retrieves batches from the bridge blockchain and submits transactions to the destination chain.

Bridging workflow

The following example demonstrates a cross-chain token transfer from Prime to Vector, where Prime is the source chain and Vector is the destination chain. A similar same process applies to transfers from Prime to Nexus or from any L2 network back to Prime.

The red-highlighted components indicate the audit scope.

1. **Submit bridging transaction**: the sender initiates the cross-chain transfer by creating a transaction on the source chain using their funds to cover the network fee for block inclusion, the bridging fee for cross-chain relay, and the amount of tokens to be bridged. The funds are locked in the bridge's multisig (hot) address, and the transaction metadata contains the destination chain ID and address, and amount.
2. **Detect bridging transaction**: the oracle of each validator of the bridge chain monitors bridging transactions by watching for transactions on the source chain that transfer funds to the bridge multisig address.
3. **Witness bridging transaction**: each oracle submits a bridging request claim to the bridge blockchain to confirm it has observed the bridging transaction.
4. **Confirm bridging transaction**: the transaction becomes valid and ready for batching once a quorum of validator oracle votes confirms it.

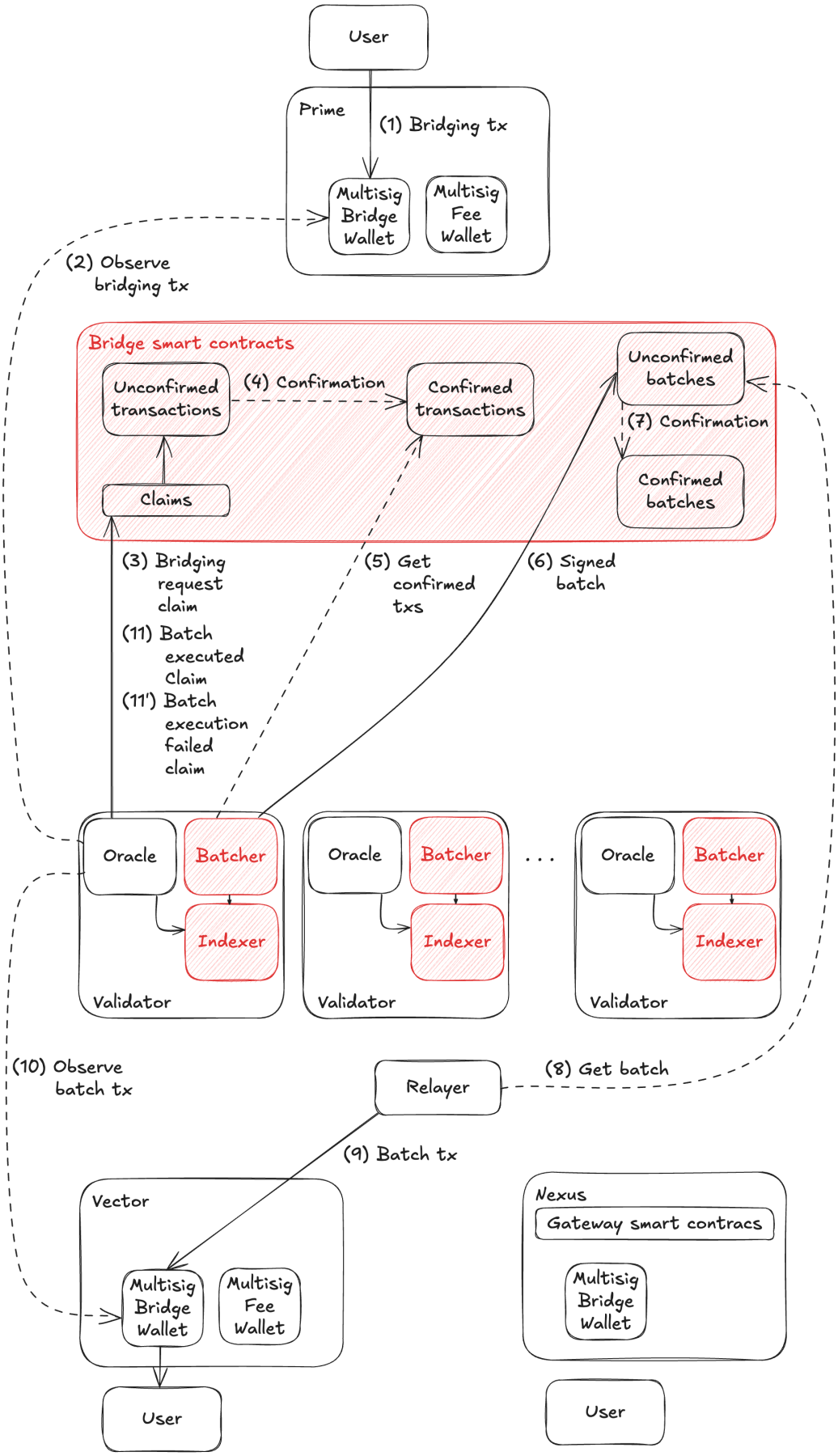

Figure 1: Reactor bridge

5. **Retrieve confirmed transactions**: the batcher generates a batch with the confirmed transactions when either enough bridging transactions have been confirmed or the maximum time between batches is reached.
6. **Submit batch transaction:** to construct the batch, the batcher leverages the block indexer's database to acquire input UTXOs that belong to the multisig address controlling the bridged funds and to the multisig address responsible for covering the network fee on the destination chain. Following that, output UTXO instances associated with receiver addresses are formed by processing the confirmed bridging transactions retrieved from the bridge.
7. **Confirm batch transaction:** before submission to the destination blockchain, the batch requires confirmation from a quorum of bridge validators.
8. **Retrieve confirmed batch**: the relayer retrieves the confirmed batch and it combines the individual signatures from the batch to create the multi-signature needed to release the specified funds on the destination chain.
9. **Submit batch**: the batch transaction is submitted to the destination blockchain. The bridged funds are unlocked from the bridge-controlled multisig address and transferred to the destination addresses.
10. **Detect batch execution**: when the batch is successfully executed on the destination blockchain, the oracle component of each validator will submit a batch executed claim.
11. **Witness batch execution**: once a quorum of signatures is collected, the batch is marked as executed on the chain. The bridge then updates its internal state, allowing it to generate the next batch containing any new confirmed transactions that were submitted during this process.

Components in scope

The Reactor bridge consists of multiple components but these audit focuses on the following:

Indexer

The block indexer for Cardano chains stores block and transaction information in a database. Its key features are:

- **Configurable confirmation depth**: it waits for a specified number of block confirmations before processing the block. Block headers are stored in a circular queue (of length equal to the number of desired confirmations), and when the queue is full, the first element from the queue is processed.
- **Transaction filtering**: it can track specific addresses and their transactions.
- **UTXO tracking:** it can process and store transaction inputs and outputs (i.e. UTXO inputs spent and outputs created) for either all transactions in a block, or only the transactions that involve any address of interest. It supports different transaction eras (Allegra, Alonzo, Babbage, Conway, Mary, Shelley).
- **Rollback handling:** it supports chain reorganizations through roll-backward functionality, being able to roll back to previous states when chain reorganizations occur.

Batcher

As mentioned before, the batcher is off-chain component run on each validator. Each instance of a batcher is used for a distinct destination, meaning there will be 3 batchers per validator: one for Prime, one for Nexus and one for Vector. The batchers are started by the batcher manager which serves as the orchestrator for the entire batching system. It handles the initialization and coordination of multiple batchers based on configuration (i.e. one batcher is created for every possible destination chain - one for each Cardano chain, Prime and Vector; one for the EVM chain Nexus). Each batcher instance is launched in its own goroutine, allowing for parallel processing of different chains. The manager is responsible for proper resource allocation and lifecycle management of all batchers.

The batcher is the core processing engine that continuously monitors for confirmed bridging transactions and groups them into batches.

It polls the bridge smart contract at configured intervals, verifies chain synchronization (i.e. queries the indexer's latest block point to compare with the bridge's last observed block and ensure that the batcher is synchronised before creating batches), retrieves confirmed transactions, and handles the batch creation process. The batcher implements retry logic for failed submissions and provides telemetry data for monitoring.

Chain operations provides the chain-specific implementation for transaction handling, with separate implementations for Cardano and EVM chains. For Cardano, it handles UTXO management, multisig operations, and transaction construction according to Cardano's requirements, while for EVM chains, it manages transaction creation according to EVM specifications. This abstraction allows the batcher to work consistently across different chains while handling chain-specific details appropriately.

Bridge smart contracts

The smart contracts handle oracle claim submissions, which undergo verification through a quorum-based approach. After validation, transactions are grouped into configurable batches and signed collectively by validators.

The central contract, `Bridge.sol`, serves as the primary coordinator. It manages the registration of different chains, configures validators, and orchestrates claim submission and processing. It directs operations to specialized contracts based on the transaction or operation type.

The `Claims.sol` contract specializes in verifying and processing different claim types—including bridging requests, batch executions, and refunds. It tracks token quantities across chains, handles batch creation timeouts, and manages transaction states and nonces.

The `SignedBatches.sol` contract handles the processing of batcher-submitted signed batches using a quorum-based approval system.

The `Validators.sol` contract oversees validator operations. It maintains a validator registry, ensures proper signature verification for both BLS and multisig schemes, and manages chain-specific validator data. It also calculates the required quorum for transaction validation.

Skyline bridge

The Apex Skyline bridge is built on the same foundation as the Reactor bridge, but it is tailored to enabling asset bridging between Prime and Cardano mainnet.

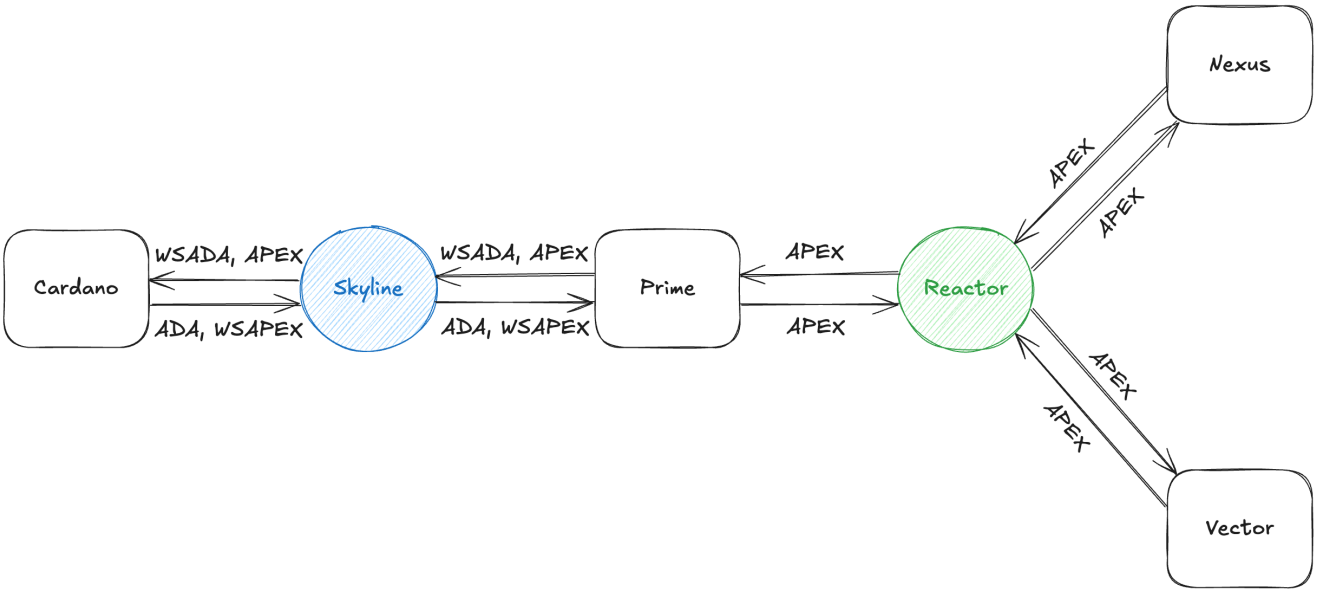

Figure 2: Skyline & Reactor bridges

Native ADA (from Cardano) and APEX (from Prime) tokens are converted into their wrapped representation when received on the destination chain (WSADA, when received on Prime; WSAPEX, when received on Cardano).

For the implementation of the Skyline bridge, there haven't been any changes in the Cardano indexer. In the batcher the main modification is that now there are not only APEX tokens in play, but also ADA which will both be wrapped. Since that is the case, changes are mainly around that new introduction:

- Getting outputs (here) from confirmed transactions is now done differently, meaning that now there is option to have tokens field as part of `TxOutputs.Outputs` array. Also, `TxOutputs.Sum` now actually has different map entries, depending on the token name.
- Getting UTXOs for normal batch has minor changes, mainly around filtering out unknown tokens (here) and calculating minimum *lovelace* amount (here) which will be used in `getNeededUtxos()`.

- `getNeededUtxos()` is quite similar, while only changes are that now instead of coding the algorithm in batcher, it is using `cardano-infrastructure/indexer` package to do so (here).
- getting UTXOs for consolidation is quite similar again, however only change is that now unknown tokens are filtered out (here).
- Deciding how outputs should look like in the created transaction is also changed a bit (here).

And for the bridge smart contracts, the main modifications are:

- The replacement of a single `totalAmount` field in the `BridgingRequestClaim` structure for four new fields, namely the amounts of native and wrapped tokens on the source chain, and the amounts of native and wrapped tokens on the destination chain. Both native (ADA, APEX) and wrapped (WSADA, WSAPEX) tokens may bridged through the Skyline bridge, so both for the source chain and the destination chain there are two distinct amounts to account for differences between source and destination amounts due to payment of fees.
- A new mapping `chainWrappedTokenQuantity` is added to the `Claims.sol` contract to keep track of the amount available of wrapped tokens on the destination chain.

Threat Model

During our threat modelling we have identified several properties and based on them we have formulated threats we have inspected. We have listed them bellow.

Threats that are related to the Reactor bridge's batcher and smart contracts also relate to Skyline's as well.

Property BSC-01: if a sender submits a bridging request, the funds must eventually be either received on the destination or refunded to the sender

Violation consequences

- Violation of this property could lead to loss of funds.

Threats

- Information of batch claim stored in the bridge is lost before the relayer can retrieve the raw transaction data and submit it on the destination chain.
- Information of the bridging transaction stored in the bridge is lost before a refund request claim is submitted for the bridging transaction. In that case the oracle would not be able to construct the refund request claim that would eventually lead to the refund of the funds.
- The source chain's multisig hot wallet lacks sufficient funds to process refunds to senders.

Conclusion

After reviewing the components under scope, the property holds under the assumptions that:

- There are trusted, honest, correctly functioning oracles and batchers, and a live, honest relayer. These components should be live and work as expected.
- There are eventually enough funds on the hot wallet of either the destination or source chain to transfer the funds to the receiver on the destination chain, or refund the sender on the source chain.

Property BSC-02: claims and batches must be processed only for registered chains

Violation consequences

- Violation of this property could lead to loss of funds. For example, if the source chain is malicious and is not registered in the bridge, a fraudulent bridging request claim might be submitted by honest oracles monitoring the unregistered chain.

Threats

- A claim is processed for an unregistered (potentially malicious) source chain, which would lead to unlocking of funds on a registered, honest destination chain.

Conclusion

The property holds:

- For bridging request claims, it is checked that both source and destination chains are registered.
- For batch executed claims, it is checked that the destination chain (where the batch executed) is registered.

- For batch execution failed claims, it is checked that the destination chain (where the batch didn't successfully execute) is registered.
- For refund request claims, it is check that the source chain (where the user should get their funds back) is registered.
- For hot wallet increment claims, it is checked that the chain (where the hot wallet was funded) is registered.
- For batches, it is check indirectly through the function `shouldCreateBatch()` that the destination chain (where the batch should execute) is registered.

Additionally, functions `defund()`, `updateChainTokenQuantity()` and `submitLastObservedBlocks()` also perform the same check.

Property BSC-03: bridging request claims can only be submitted by trusted oracles

Violation consequences

- Violation of this property could lead to loss of funds, as fraudulent bridging request claims would be accepted. Once quorum is reached for the bridging request, a batch containing these claims would execute on the destination chain—allowing withdrawals from the multisig wallet without corresponding locked funds on the origin chain.

Threats

- If enough malicious actors reach quorum, they can submit fraudulent bridging request claims—claims not based on legitimate transactions on origin chains—and drain funds from multisig wallets on honest destination chains.

Conclusion

The property holds. The presence of the `onlyValidator` modifier in function `submitClaims()` guarantees that the sender of the message must be a trusted oracle signing the message with the private key of its corresponding validator (of the bridge). Function `submitClaims()` is the only externally-accessible entry point for claims submission.

Property BSC-04: a bridging request claim will be processed if and only if a quorum of validators have signed over it

Violation consequences

- Violating this property could lead to loss of funds. Without a quorum requirement, a single malicious (trusted) oracle could submit a fraudulent bridging request claim—that is, a claim for a non-existent bridging transaction on the source chain. If such a claim is processed and included in a batch, it would allow unauthorized unlocking of funds from the destination chain's multisig wallet, even though no funds were actually locked in the source chain's multisig wallet.

Threats

- A bridging request claim that has not collected votes from a quorum of validators is confirmed and triggers a refund on the source chain.

Conclusion

The property holds. The claim is processed only if quorum is reached.

Property BSC-05: every confirmed bridging request claim must eventually enter a batch

Violation consequences

- Violation of this property could lead to liveness issues. When a confirmed bridging request claim reaches quorum but remains excluded from a batch, the sender's funds cannot reach their destination. In such cases, the sender must initiate a refund process to recover their funds.
- Violation of this property could lead to loss of funds if oracles fail to reach quorum, or if the information of the confirmed bridging request is lost.

Threats

- Information of bridging request claim stored in the bridge is lost before the batcher can retrieve it to generate a batch.
- A quorum of oracles are not able to reach agreement on the bridging request claim for the bridging transaction of the user.

Conclusion

Assuming that batchers eventually reach agreement on batches and that a quorum of batchers are online, then this property holds. When a bridging request claim becomes confirmed after reaching quorum, then the claim is added to set of confirmed transactions. When the next batch (or one of the following batches) needs to be created the claim will be retrieved by the batchers through the `getConfirmedTransactions()` query. There is no code that deletes entries from the `confirmedTransactions` mapping, so it is not possible that entries get accidentally deleted. The claim should be processed by the batchers, who should eventually reach agreement on the batch.

Property BSC-06: no confirmed bridging request claim is included in more than one batch

Violation consequences

- Violation of this property could lead to loss of funds if a confirmed bridging request claim is included in multiple batches that execute on the destination chain—the receiver(s) would receive funds more than once.

Threats

- A confirmed bridging request claim is included in more than one batch.

Conclusion

The property holds. After a batch has either executed successfully or failed/timed out on the destination chain, oracles must submit (and agree on) a batch executed claim or batch execution failed claim, respectively:

- If a batch executed claim is confirmed, then the `lastBatchedTxNonce` value for the destination chain is updated with the nonce of the last transaction that was included in the batch (see here).
- If a batch execution failed is confirmed, then the `lasBatchedTxNonce` value is also updated with the nonce of the last transaction that was included in the batch (see here).

When a new batch needs to be create, the nonce of the first transaction to be part of the set of confirmed transactions that will make it into the batch is `lastBatchedTxNonce + 1` (see here). And since for every destination chain only one in-flight batch is allowed (see Property BSC-15), then it is not possible that a confirmed transaction might be included in two concurrent batches.

Property BSC-07: confirmed bridging request claims remain stored on the bridge until a batch transaction either succeeds or fails on the destination chain

Violation consequences

- Violation of this property could lead to safety issues, because it would not be possible to refund users on the origin chain if the batch transaction on the destination fails or does not succeed before the timeout expires.

Threats

- Confirmed bridging requests are not stored in contract storage.
- Confirmed bridging requests are removed from contract storage before the bridging transaction lifecycle completes.

Conclusion

The property holds. When validators reach quorum on a bridging request claim, it gets stored in the `confirmedTransactions` mapping. This mapping uses the destination chain ID and transaction nonce assigned to the claim as indices. The confirmed transaction remains in storage indefinitely, as it is never removed from the `confirmedTransactions` mapping—even after the batch relaying the transaction succeeds or fails on the destination. When either a batch executed claim or a batch execution failed claim is submitted, the transaction stays in the mapping. The `lastBatchedTxNonce` value for the destination chain is then updated with the transaction nonce of the last transaction in the batch, which determines the nonce for the first transaction in the next batch (here and here).

Property BSC-08: batches can only be submitted by trusted batchers

Violation consequences

- Violation of this property could lead to loss of funds if fraudulent batches are accepted. When malicious batches reach quorum, they would execute on the destination chain—allowing withdrawals from the multisig wallet without having corresponding locked funds on the origin chain.

Threats

- If enough malicious actors reach quorum, they can submit fraudulent batches (i.e., claims not based on legitimate bridging request transactions on origin chains) to drain funds from multisig wallets on honest destination chains.

Conclusion

The property holds. The presence of the `onlyValidator` modifier in functions `submitSignedBatch()` (here) and `submitSignedBatchEVM()` (here) functions guarantees that the sender of the message must be a trusted batcher signing the message with the private key of its corresponding validator (of the bridge). These two function are the only externally-accessible entry points for batch submission.

Property BSC-09: batch executed claims can only be submitted by trusted oracles

Violation consequences

- Violation of this property could lead to liveness issues if fraudulent batch executed claims are accepted. When malicious malicious batch executed claims reach quorum before funds are received on the destination chain, batchers might agree on the next batch prematurely. This would prevent the transfer of funds in the previous batch from completing, requiring users to initiate a refund process to recover their funds.

Threats

- If enough malicious actors reach quorum, they can submit fraudulent batch executed claims before the batch has actually succeeded or failed on the destination chain.

Conclusion

The property holds. The presence of the `onlyValidator` modifier in function `submitClaims()` guarantees that the sender of the message must be a trusted oracle signing the message with the private key of its corresponding validator (of the bridge). Function `submitClaims()` is the only externally-accessible entry point for claims submission.

Property BSC-10: failed batch claims can only be submitted by trusted oracles

Violation consequences

- Violation of this property could lead to double spend through fraudulent batch execution failure claims. If funds have been successfully received on the destination chain but a malicious batch execution failed claim is processed, it would trigger an unlocking of funds on the source chain and refund them to users—resulting in funds existing simultaneously with both senders and receivers.

Threats

- If enough malicious oracles reach quorum, they can submit a batch execution failed claim after the batch has actually succeeded on the destination chain.

Conclusion

The property holds. The presence of the `onlyValidator` modifier in function `submitClaims()` guarantees that the sender of the message must be a trusted oracle signing the message with the private key of its corresponding validator (of the bridge). Function `submitClaims()` is the only externally-accessible entry point for claims submission.

Property BSC-11: for any given batch, all batchers must include the same set of confirmed transactions

Violation consequences

- Violation of this property could lead to liveness issues, as validators would fail to reach consensus on a batch transaction. This would prevent the batch transaction from being relayed to the destination chain, leaving transferred funds blocked.

Threats

- When batchers request the set of confirmed transactions for a given batch ID, they receive different sets, preventing them from generating a batch that can achieve quorum.

Conclusion

The property holds. When a new batch needs to be generated batchers query the bridge smart contract to retrieve the confirmed transactions by calling `getConfirmedTransactions()`. By the time a new batch needs to be generated, the set of confirmed transactions included in the batch is fixed, and determined by the nonce of the first transaction in the batch (calculated by incrementing by 1 the nonce of the last transaction included in the previous batch) and the number of transactions to include. For any given batch, all batchers will retrieve the same set of transactions.

Property BSC-12: a batch will never be relayed to the destination if a quorum of validators have not signed over it

Violation consequences

- Violation of this property could lead to loss of funds. If there's no quorum requirement, a single malicious validator could sign and relay a batch to the destination chain, enabling unauthorized withdrawals from the multisig wallet on the source chain.

Threats

- A batch becomes confirmed without a quorum of validators voting for it.

Conclusion

The property holds. The `lastConfirmedBatch` mapping is only updated with a confirmed batch for any given destination chain when quorum has been reached for the signed batch (see here). Unless the batch is stored in the mapping, the batch will not be relayed.

Property BSC-13: a batch will eventually be relayed if and only if a quorum of validators have signed over it

Violation consequences

- Violation of this property could lead to liveness issues. When a batch reaches quorum but isn't relayed by the relayer, the cross-chain transfer cannot complete. If the batch isn't relayed before the transaction timeout, an oracle must submit a batch execution failed claim.

Threats

- The relayer is offline or cannot relay the batch.

Conclusion

The property holds. Assuming there is an honest, correctly functioning, online relayer, the batch stored in the `lastConfirmedBatch` mapping for any given destination chain will be relayed.

Property BSC-14: for every confirmed batch, the bridge must eventually process either a batch executed claim or a batch execution failed claim

Violation consequences

- Violation of this property could lead to liveness issues. Without either a batch executed claim or a batch execution failed claim being processed, the bridge would stop allowing cross-chain transfers to the destination chain since it cannot generate new batches for that destination chain.

Threats

- Oracles are unable to submit either batch executed claims or batch execution failed claims.
- Oracles cannot reach consensus on the batch executed claim or batch execution failed claim for a given batch.
- The processing of batch executed claims or batch execution failed claims fails during transaction execution.

Conclusion

The property holds based on the assumption of honest, correctly functioning relayer and oracle components.

Property BSC-15: there must be at most 1 in-flight batch per destination chain

Violation consequences

- Violation of this property could lead to liveness issues. If a new batch is generated and confirmed while the previous batch hasn't completed its lifecycle, transactions in the previous batch could get stuck. This would require manual intervention to either refund users on the source chain or relay transactions to the destination chain (if the timeout hasn't expired).

Threats

- A new batch is confirmed by a quorum of validators before the previous one has completed its lifecycle.

Conclusion

The property holds. When a signed batch is confirmed, it is stored in the `confirmedSignedBatches` mapping. Additionally, the `currentBatchBlock` mapping for the destination chain is set to the current block height of the bridge. By setting `currentBatchBlock` to an unsigned integer greater than 0, the bridge is signalling that the confirmed batch needs to be relayed and complete its lifecycle, because any calls to the `getNextBatchId()` query will return a zero value (`currentBatchBlock` for the given destination chain is non-zero, `shouldCreateBatch()` returns false and then `getNextBatchId()` returns 0). When the batch's lifecycle completes, and oracles agree on either a batch executed claim or a batch execution failed claim, then the `currentBatchBlock` for the corresponding destination chain will be reset to -1 (here and here), and only then `getNextBatchId()` might return a non-zero value if the conditions are met to generate a new batch (the number of confirmed transactions has met the value of `maxNumberOfTransactions` or the block height of the bridge has reach the `nextTimeoutBlock` and there confirmed transactions to batch).

Property BSC-16: a batch execution failed claim will be processed if and only if a quorum of validators have signed over it

Violation consequences

- Violation of this property could lead to loss of funds. Without a quorum requirement, a single malicious (trusted) oracle could submit a fraudulent batch execution failed claim. If this claim is processed, it would allow unauthorized unlocking of funds from the multisig wallet on the source chain—and potentially simultaneous withdrawal of funds from the multisig wallet on the destination chain if the relayer successfully executes also the batch transaction on the destination.

Threats

- A batch execution failed claim that has not collected votes from a quorum of validators is confirmed and triggers a refund on the source chain.

Conclusion

The property holds. The claim is processed only if quorum is reached.

Property BSC-17: for any batch that does not successfully execute on the destination chain, at most one failed batch claim must be processed

Violation consequences

- Violation of this property could lead to liveness issues. Processing multiple batch execution failed claims for the same batch would cause incorrect adjustments to the multisig wallet's internal fund tracking, making it inconsistent with the actual balance on the destination chain.

Threats

- More than one batch execution failed claim for the same batch is processed and confirmed by a quorum of validators.

Conclusion

The property does not hold. A finding has been reported where we show that it is possible that two batch execution failed claims (with same batch ID, but different observed transaction hashes) are processed after reaching quorum.

Property BSC-18: refund request claims can only be submitted by trusted oracles

Violation consequences

- Violation of this property could lead to double spend through fraudulent refund request claims. If funds have been successfully received on the destination chain but a malicious refund request claim is processed, it would trigger an unlocking of funds on the source chain and refund them to users—resulting in funds existing simultaneously with both senders and receivers.

Threats

- If enough malicious oracles reach quorum, they can submit a refund request claim after the batch has actually succeeded on the destination chain.

Conclusion

The property holds. The presence of the `onlyValidator` modifier in function `submitClaims()` guarantees that the sender of the message must be a trusted oracle signing the message with the private key of its corresponding validator (of the bridge). Function `submitClaims()` is the only externally-accessible entry point for claims submission.

Property BSC-19: a refund request claim will be processed if and only if a quorum of validators have signed over it

Violation consequences

- Violation of this property could lead to loss of funds. Without a quorum requirement, a single malicious (trusted) oracle could submit a fraudulent refund request claim. If this claim is processed, it would allow unauthorized unlocking of funds from the multisig wallet on the source chain—and potentially simultaneous withdrawal of funds from the multisig wallet on the destination chain if the relayer successfully executes also the batch transaction on the destination.

Threats

- A refund request claim that has not collected votes from a quorum of validators is confirmed and triggers a refund on the source chain.

Conclusion

The property holds. The claim is processed only if quorum is reached.

Property BSC-20: for any batch that does not successfully execute on the destination chain, at most one refund request claim must be processed for each bridging request included in the batch

Findings: Quorum may be reached for multiple refund request claims that only differ on one field

Violation consequences

- Violation of this property could lead to loss of funds. If multiple refund request claims for a single bridging request are processed, this would result in duplicate refunds from the multisig wallet on the source chain.
- Transactions included in a batch for UTXO chain have attached TTL (Time To Live) which could lead to transactions timeout on the destination chain. If this is the case, refund request should be submitted by the oracle.

Threats

- A refund request claim that has not collected votes from a quorum of validators is confirmed and triggers a refund on the source chain.
- Users will not lose funds if, due to TTL property of UTXO, transactions timeout on the destination chain

Conclusion

The property does not hold. A finding has been reported where we show that it is possible that two refund request claims that differ on only one (unused at the moment) field are processed after reaching quorum.

Property BSC-21: hot wallet increment claims can only be submitted by trusted oracles

Violation consequences

- Violation of this property could lead to liveness issues through fraudulent hot wallet increment claims. If the internal tracking of funds in the multisig wallet is artificially inflated and does not match the actual balance, the bridge would accept transfer requests exceeding available funds, causing transactions to fail on the destination chain.

Threats

- A malicious actor submits a hot wallet increment claim or triggers an increment through other means, artificially manipulating the internal fund tracking of the multisig wallet for any chain.

Conclusion

The property holds. The presence of the `onlyValidator` modifier in function `submitClaims()` guarantees that the sender of the message must be a trusted oracle signing the message with the private key of its corresponding validator (of the bridge). Function `submitClaims()` is the only externally-accessible entry point for claims submission.

Property BSC-22: a hot wallet increment claim will be processed if and only if a quorum of validators have signed over it

Violation consequences

- Violation of this property could lead to liveness issues through fraudulent hot wallet increment claims. Without a quorum requirement, a single malicious oracle could submit a fraudulent hot wallet increment claim.
- If the internal tracking of funds in the multisig wallet is artificially inflated and does not match the actual balance, the bridge would accept transfer requests exceeding available funds, causing transactions to fail on the destination chain.

Threats

- A malicious actor submits a hot wallet increment claim or triggers an increment through other means, artificially manipulating the internal fund tracking of the multisig wallet for any chain.

Conclusion

The property holds. The claim is processed only if quorum is reached.

Property BSC-23: only the fund admin is authorized to execute a hot wallet defund transaction

Violation consequences

- Violation of this property could lead to loss of funds—if unauthorized actors can execute a hot wallet defund transaction, they could drain funds from the multisig account on the destination chains.

Threats

- Unauthorized actors could submit and successfully execute a hot wallet defund transaction, allowing it to be confirmed and potentially included in a batch for execution on the destination chain.

Conclusion

The property holds. The `defund()` function is protected by the `onlyAdminContract` modifier, which checks whether the message sender matches the address of admin contract.

Property BSC-24: the bridge blockchain must accurately track the quantity of tokens available in the multisig accounts on all registered chains

Violation consequences

- Violation of this property could lead to liveness issues because if the tracking of the multisig wallet on chains is incorrect (especially if it shows less than the actual balance), the bridge might reject transactions when the transfer amount exceeds the tracked available funds on the destination chain.

Threats

- The multisig wallet balances tracked by the bridge can be modified without corresponding actual changes to the wallet balances on their respective chains.

Conclusion

The property holds.

Reactor bridge

For every chain connected to the Reactor bridge, the mapping `chainTokenQuantity` keeps track of the amount of native tokens available for transfer. During a cross chain transfer the value, for any given chain, is updated as follows:

- When a bridging request claim is processed, the available amount on the source chain is increased, and the available amount on the destination chain is decreased (both by the total amount of all bridging transfers in the claim).
- When a batch executed claim is processed, no updates need to be done.
- When a batch does not executes successfully on the destination chain, the updates done when handling the bridging request claim must be reverted:
  - First, when a batch execution failed claim is processed, the available amount on the destination chain is increased, since the tokens were not sent to the recipients on the destination chain and the tokens are thus still available on the destination chain.
  - Second, when a refund request claim is processed, the available amount on the source chain is decreased, since the tokens need to be refunded to the senders of the bridging transactions included in the batch.

Additionally:

- When processing a hot wallet increment claim, the available amount on the chain is increased, since the balance of the multisig wallet on the chain should have increased.
- When processing a defund request, the available amount on the chain is decreased, since the balance of the hot multisig wallet will be decreased as a result of transferring funds to the cold multisig wallet, once the request is included in a batch, and the batch executes successfully on the chain.
- When the function `updateChainTokenQuantity()` is called, the available amount on the chain either increases or decreases, depending on the `_isIncrease` parameter of the function.

Skyline bridge

Furthermore, for the Skyline bridge, the mapping `chainWrappedTokenQuantity` keeps track of the amount of wrapped tokens available for transfer, and during a cross chain transfer the value, for any given chain, is updated as follows:

- When a bridging request claim is processed, the available amount on the source chain is increased, and the available amount on the destination chain is decreased (both by the total amount of all bridging transfers in the claim).
- When a batch executed claim is processed, no updates need to be done.
- When a batch does not executes successfully on the destination chain, the updates done when handling the bridging request claim must be reverted:
  - First, when a batch execution failed claim is processed, the available amount on the destination chain is increased, since the tokens were not sent to the recipients on the destination chain and the tokens are thus still available on the destination chain.
  - Second, when a refund request claim is processed, the available amount on the source chain is decreased, since the tokens need to be refunded to the senders of the bridging transactions included in the batch.

Additionally:

- When processing a hot wallet increment claim, the available amount on the chain is increased, since the balance of the multisig wallet on the chain should have increased.
- When processing a defund request, the available amount on the chain is decreased, since the balance of the hot multisig wallet will be decreased as a result of transferring funds to the cold multisig wallet, once the request is included in a batch, and the batch executes successfully on the chain.
- When the function `updateChainWrappedTokenQuantity()` is called, the available amount on the chain either increases or decreases, depending on the `_isIncrease` parameter of the function.

Property BTCHR-1: lagging behind batcher will eventually catch up with the rest of the batchers

Findings: Out-of-sync batchers are not able to sync with others

Violation consequences

- Violation of this property could lead to liveness issues. If batcher come out of sync, it should be able to pick up with the rest of the batchers. Synchronization is defined by batcher's indexer having information about destination which is newer or the same as the smart contract has. If batcher is not able to get newer blocks from the indexer, that batcher will never generate and sign a batch which can have critical consequences eventually halting the bridge if enough batchers simultaneously fail.

Threats

- The lagging behind batcher is not able to sync up.

Conclusion

Based on the threat inspection, we have concluded that if the batcher's indexer is not synched with the others, it will provide the batcher with the outdated data constantly, meaning that the batcher will never be able to sync up with the others, since there is no mechanism implemented on the batcher's side, which would guarantee that the indexers and oracles know that they are well behind and that they should sync with others.

Property BTCHR-2: batchers having out-of-sync indexers can generate valid batches

Violation consequences

- Violation of this property can lead to liveness issues. Every validator runs a local batcher and indexer processes, and indexers will query different nodes of a destination chain. The system should not assume that all batchers

will have the same view of the UTXOs and block slots on the destination chain, so there should be some kind of buffer which will enable batchers to agree on the batch, since UTXOs and block slot are a part of transaction hash.

Threats

- Out-of-sync batchers can halt the batch quorum.
- TTL property of UTXO transactions causes batches to have different hashes.

Conclusion

Out-of-sync batchers can halt the batch quorum

- Inspecting this threat we have concluded that if a batcher is out of sync with the bridge smart contract, this check here will disable it from submitting batches that would be outdated. However, if some batchers have different view of the indexer's data, due to asynchrony, there is a possibility that batchers will have different source of UTXOs to create Cardano transaction. However, similar scenario is outlined in the "Property BTCHR-3: batchers are creating batches in a deterministic way".
- Assume that batcher `b` runs indexer `i` which is lagging behind others (but is still deemed valid according to the mentioned check against the smart contract) and supplies batcher with outdated data. Assume that other batchers have a view of the destination data which is different (newer) than the batcher `b`.
  - If a Cardano batcher `b` submits batch `btch` with UTXOs which are not available to the other indexers, it will cast a vote for a hash `h` . Assume that other validator's Cardano batchers submit a batch `b'` which does not contain some of the UTXOs from the `btch` and eventually they start agreeing on the content of the batch. Cardano batcher `b`'s indexer will ingest new state of the system and eventually will reason that some of the UTXOs from the `btch` are not available anymore and therefore create a new batch with new hash. Since other batchers have faithfully submitted a batch with adequate UTXO content, they will not provide a new batch to the contract, leaving only batcher `b` to post the same batch once its indexer pick up new data.

TTL property of UTXO transactions causes batches to have different hashes

- Inspecting this threat lead us to inspect rounding functions which will be called on top of the block slots (here) and block numbers respectively (here). TTL is integral part of Cardano and Ethereum transactions and will be a part of the tx hash. These functions ensure that if the differences in the indexers' views of the chain data is different, they would still be able to reach agreement about the batch, since minor differences between block slots (or block numbers) will yield the same rounded value, ensuring that TTL will have no impact on tx hash difference.

Property BTCHR-3: batchers are creating batches in a deterministic way

Violation consequences

Violation of this property can lead to liveness issues. If batchers are creating batches in nondeterministic way, they may never reach quorum, since the hashes will be different.

Threats

- UTXOs are chosen in nondeterministic way and prevent quorum from being reached.
- Constant funding of hot wallet on the destination chain, by malicious actors, prevents batch from being created in the deterministic way.

Conclusions

Constant funding of hot wallet on the destination chain, by malicious actors, prevents batch from being created in the deterministic way

- There is a cap on the maximum number of UTXOs that can be in the transaction, meaning that any batcher will create the same Cardano transaction using the maximum number of UTXOs in the transaction. Eventually, all batchers will agree on the same since even if some of the transactions are exchanged (here), eventually all batchers' indexers will have the same view of the UTXOs available on the destination chain.
- Another possible flaw here is the number of fee UTXOs that could be included in the batch, since different indexers can have different view of the fee multisg account. However, there is a cap on the maximum number of fee UTXOs that can be included in the batch, meaning that even if the malicious actor funds fee multisig address, eventually maximum number of fee UTXOs will be selected in ordered manner.

UTXOs are chosen in nondeterministic way and prevent quorum from being reached

- UTOX are received from the indexer in an ordered way. We have already reported the critical finding around sorting transactions. If we assume that the sorting bug is fixed, then we can assume that the UTXOs will be chosen in the deterministic manner, since the `getNeededUtxos()` function will operate on the same array of transactions.

Property BTCHR-4: high frequency of bridging requests will not cause system to halt

Violation consequences

- Violation of this property could lead to liveness issues. There has to be a mechanism which in the event of a large amount of bridging requests towards a certain destination chain, the bridging will not be significantly slowed down or halted by batchers. Batchers should be able to agree upon a batch, even in this scenario.

Threats

- Significantly slowed down bridging due to high or low frequency of bridging requests.
- Bridging is halted due to high frequency of bridging requests because batchers cannot agree on the batch.

Conclusions

Significantly slowed down bridging due to high or low frequency of bridging requests

Based on the threat inspection we have concluded that there will be no significant delay apart from the one dictated by the protocol's design. Protocol design dictates that **eventually** components will reach an agreement about certain information such as batch content, bridge claims, etc.

- Smart contracts implement timeout mechanism which in events of low frequency will trigger batch creation. This inserts possible delay in the bridging, but it is accounted for in the protocol.
- Smart contracts implement a maximum number of confirmed transactions, meaning that in high bridging request frequency scenarios, eventually batchers will have the same set of confirmed transactions for which they will create a batch.

Bridging is halted due to high frequency of bridging requests because batchers cannot agree on the batch

Based on the threat inspection we have concluded that honest batchers will eventually generate a batch on which they will be able to reach quorum. Smart contracts store value for the maximum number of transactions that should be included in the batch, meaning that if there is a lot of transactions submitted to the batch, they will be capped at the max number, allowing batchers to eventually get the same set of transactions from the smart contracts.

Property BTCHR-5: batcher must be resilient to Cardano connection failures

Findings: The system heavily depends on external Gouroboros library

Violation consequences

Violation of this property leads to liveness issues. In order to create a batch, batcher requires protocol parameters gotten by Gouroboros protocol. To get them, batcher connects to Gouroboros protocol and it gets protcol parameters from the Cardano chain. If this connection fails, Cardano transaction cannot be created and the batcher will return error.

Threats

- Error in connecting to Gouroboros protocol halts batch production destined for UTXO chains

Conclusion

Based on performed threat inspection we have concluded that the Cardano related batcher are not resilient to errors and timeouts when connecting to Gouroboros protocol. Based on this, we have formulated a finding.

Property CNS: quorum is reached when more than two thirds of validator agree on a decision

Findings: Calculated quorum may be lower than required to guarantee BFT, Incorrect quorum formula fails to guarantee BFT safety in edge cases

Violation consequences

- Violation of this property leads to safety issues. Distributed system in which the quorum could be reached with 2/3 of the votes is not resistant to Byzantine faults.

Threats

- Formulas for calculating quorum do not faithfully represent theoretical concepts

Conclusion

The implementation does not ensure the property. Formulas used to decide on the minimum amount of signers for a particular batch do not represent theoretical concepts. Based on performed threat inspection, we have reported two findings.

Property IDX-01: all validators must set the same address configurations in their block indexer

Findings: Validators have different address configurations in their block indexer

Violation consequences

- Violation of this property could lead to liveness issues, as validators would fail to reach consensus on the observed bridge claims.

Threats

- One or more validators have mismatched address configurations in their indexer. As a result, they filter and store different UTXO and transaction sets, preventing the validators from reaching quorum on bridge claims and batches, and halting the bridge.

Conclusion

Property IDX-02: indexers must gracefully handle rollback in case of chain reorganizations

Violation consequences

- Violation of this property could lead to liveness problems, as any validator that has synchronised blocks resulting from a fork cannot recover from it.

Threats

- Validators cannot rollback incorrect blocks after being informed of a chain reorganization and halt.

Conclusion

- The indexer maintains a circular queue of unconfirmed blocks and ensures that all blocks from this queue have gone through the required confirmation depth before being processed.
- When a chain reorganization is detected by the indexer's Cardano node, a rollback to the latest confirmed block point in the queue is attempted. If successful, all blocks are truncated until that point; otherwise, the validator halts.
- It's therefore important that all validator indexers are configured with sufficient queue length for chain reorganisation. The block time of the chain and the historical reorganisations analysis should be used to determine the queue length.

Property IDX-03: indexers must perform database operations atomically to ensure consistency

Violation consequences

- Violation of this property could lead to liveness issues. If a node crashes during a write operation, the database may reflect a partially processed block. This would break determinism, leading to inconsistent state across validators, preventing validators to reach quorum on observed bridge claims and batches.

Threats

- Validators may operate on or attest an inconsistent view of the blocks.

Conclusion

For each block processed, indexers store the resulting UTXOs, transactions and block information in one atomic transaction.

Property IDX-04: indexers must be resilient to Cardano connection failures

Violation consequences

- Violation of this property could lead to liveness issues. If a node indexer loses its connection to its Cardano node, it will stop synchronizing new blocks. This will prevent the node from participating in the validators quorum and could prevent the bridge from operating.

Threats

- Indexers stop synchronizing new blocks and halt the oracle and batch quorum.

Conclusion

- The indexer implementation has a restart mechanism in case of non-fatal errors.
- The indexer runs a dedicated Go routine to detect connection failures to its Cardano node.

Property IDX-05: indexers must maintain a single connection to a Cardano node

Violation consequences

- Violation of this property could lead to liveness and safety issues. If an indexer establishes **multiple concurrent connections** to a Cardano node, the confirmation queue may be populated in parallel by conflicting data streams. This can result in **unordered or duplicated blocks** entering the queue.

  Since the indexer's rollback and restart logic assumes that blocks in the queue are **monotonically ordered and free of duplicates**, any deviation from this assumption during a node restart or chain reorganization can lead to serious inconsistencies. Specifically:

  - **Invalid blocks may be confirmed and processed**, breaking consistency guarantees.
  - **Valid blocks may be dropped**, preventing the system from detecting real transactions.

    These inconsistencies can compromise both the **safety** (e.g., incorrect attestations) and **liveness** (e.g., bridge halting) of the bridge system.

Threats

- An Indexer establishes multiple connections to a Cardano node and its confirmation queue becomes unordered due to parallel block ingestion. Later a rollback or node restart occur, and invalid blocks are confirmed or valid blocks may be dropped. This leads to corrupted local state, missed transactions, or bridge claims compromising bridge correctness and halting processing.

Conclusion

- The block syncer of the indexer employs a locking mechanism to ensure that only once active connection to the Cardano node is maintained at any given time.
- Both roll-forward and roll-backward operations are protected by a mutex, ensuring thread-safe access to the confirmation queue and preserving its integrity.

Property IDX-06: indexers must maintain UTXO states of the observed chains

Findings: Unreliable sorting of UTXOs in database query

Violation consequences

- Violation of this property could lead to liveness issues since batchers need UTXOs to generate batches. If indexers do not maintain both new and spent UTXOs on the Cardano chains in their database in a deterministic way across all indexers, it can disrupt batcher quorum and halt batch generation.

Threats

- Initial UTXO Storage: Indexers fail to properly record new UTXOs when first observing them.
- UTXO State Management: Indexers fail to track the lifecycle of UTXOs (creation, spending, confirmation).
- Consistency Issues: Different indexers maintain inconsistent UTXO states, breaking determinism.

Conclusion

While the indexer correctly processes each transaction of interest and updates both newly created and spent UTXOs in the database, the logic of the query used to retrieve UTXOs, introduces consistency issues that breaks determinism. Based on our threat inspection, this behavior led to a reported finding titled "Unreliable sorting of UTXOs in database query".

Property IDX-07: indexers must store transactions of interest consistently in their database

Violation consequences

- Violation of this property could lead to liveness issues since the oracles must submit identical bridge claims to the bridge. If transactions containing bridge claims are not stored in a deterministic way by all indexers, the oracles quorum will fail.

Threats

- The storing of transactions across indexers is not consistent.

Conclusion

- The indexer's logic that processes and persists transactions of interest in the database is deterministic.

Findings

Finding	Type	Severity	Status
Calculated quorum may be lower than required to guarantee BFT	Protocol	Critical	Patched Without Reaudit
Unreliable sorting of UTXOs in database query	Implementation	Critical	Patched Without Reaudit
Incorrect quorum formula fails to guarantee BFT safety in edge cases	Protocol	Critical	Patched Without Reaudit
Centralisation risk due to owner with privileged rights	Implementation	High	Acknowledged
The system heavily depends on external Gouroboros library	Implementation	High	Patched Without Reaudit
Quorum may be reached for multiple refund request claims that only differ on one field	Implementation	Medium	Patched Without Reaudit
Out-of-sync batchers are not able to sync with others	Protocol	Medium	Acknowledged
Lack of validation for validators' verifying key when owner registers a chain	Implementation	Medium	Patched Without Reaudit
Quorum may be reached for multiple batch executed claims or multiple batch execution failed claims (or both) for the same batch	Implementation	Medium	Patched Without Reaudit
Addresses expected to be contract accounts may be set to EOA addresses	Implementation	Medium	Patched Without Reaudit
Contracts may be initialised with zero addresses for owner and upgrade admin	Implementation	Medium	Patched Without Reaudit
Batcher heavy relies on out of scope packages	Implementation	Medium	Acknowledged
Missed opportunity to consolidate UTXOs	Implementation	Low	Patched Without Reaudit
Loops over unbounded arrays	Implementation	Low	Patched Without Reaudit

Finding	Type	Severity	Status
Duplicate validator addresses may inflate quorum requirement	Implementation	Low	Patched Without Reaudit
Redundancy in GenerateBatchTransaction code	Implementation	Low	Patched Without Reaudit
Validators have different address configurations in their block indexer	Implementation	Informational	Acknowledged
Cardano transactions with zero amount will be created when there is no multisig and fee UTXOs	Implementation	Informational	Patched Without Reaudit
Skyline Batcher miscellaneous code concerns	Implementation	Informational	Acknowledged
Token type not specified in NotEnoughFunds event	Implementation	Informational	Patched Without Reaudit
Batchers could appear synced if smart contract has outdated block data	Implementation	Informational	Acknowledged
Bridge smart contracts miscellaneous code improvements	Implementation	Informational	Patched Without Reaudit
Optimization in validator data key verification	Implementation	Informational	Patched Without Reaudit
LevelDB indexer implementation is not tested	Implementation	Informational	Patched Without Reaudit
Redundant database queries in the processing of confirmed blocks can be optimized	Implementation	Informational	Acknowledged
Recommendations for optimising gas usage	Implementation	Informational	Patched Without Reaudit
Error handling function not fully tested	Implementation	Informational	Acknowledged
Exposing a testing function in production code	Implementation	Informational	Acknowledged
Batcher miscellaneous code concerns	Implementation	Informational	Acknowledged
Systematic lack of documentation	Documentation	Informational	Acknowledged

Calculated quorum may be lower than required to guarantee BFT

ID	IF-FINDING-001
Severity	Critical
Impact	3 - High
Exploitability	3 - High
Type	Protocol
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts/Validators.sol

Description

Processing of claims, batches and updating blocks is only allowed when validators of the bridge reach consensus (i.e., a quorum of validators have voted on the messages). The value of the quorum is calculated in function `getQuorumNumberOfValidators()`:

```
function getQuorumNumberOfValidators() external view returns (uint8 _quorum) {
  // return (validatorsCount * 2) / 3 + ((validatorsCount * 2) % 3 == 0 ? 0 : 1); is same as (A
  ↪  + B - 1) / B
  assembly {
    _quorum := div(add(mul(sload(validatorsCount.slot), 2), 2), 3)
  }
  return _quorum;
}
```

Problem Scenarios

BFT algorithms require that less than 1/3 of nodes are Byzantine, and thus more than 2/3 of nodes should behave honestly. The value returned by function `getQuorumNumberOfValidators()` does not satisfy this condition, since the calculated result for values of `validatorsCount` that satisfy the condition `((validatorsCount*2) % 3 == 0` is smaller or equal than 2/3 of validator nodes. For example, for `6` (or `9`, `12`, `15`, etc) validators, the function will return exactly `(validatorsCount*2)/3` meaning that the quorum could be reached with less than `(validatorsCount*2)/3 + 1` votes. This will happen due to rounding, since `uint` division will always round down.

Recommendation

Implement the function in such a way that it matches the `2/3 + 1` of validator nodes:

```
function getQuorumNumberOfValidators() external view returns (uint8 _quorum) {
  unchecked {
    return (validatorsCount * 2) / 3 + 1;
  }
}
```

Unreliable sorting of UTXOs in database query

ID	IF-FINDING-002
Severity	Critical
Impact	3 - High
Exploitability	3 - High
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- cardano-infrastructure/indexer/db/bbolt/bbolt.go

Description

To generate a batch, each batcher must retrieve the available UTXOs on the destination chain from their indexer database. The indexer integrates two database solutions—BoltDB and LevelDB—which implement the `GetAllTxOutputs` query. This query retrieves UTXOs for a given address, specifically the bridge multisig address on the destination chain.

When validators vote for a batch, it is identified by the hash digest of its data, including the UTXOs. To reach quorum for a defined batch, all voting validators must sign an **identical** batch. This means the UTXOs returned to the batchers by the `GetAllTxOutputs` query must be sorted.

Below, is a snippet of the sorting implementation (here), which is the same for both DB solutions.

```
sort.Slice(result, func(i, j int) bool {
  return result[i].Output.Slot < result[j].Output.Slot ||
    result[i].Output.Slot == result[j].Output.Slot &&
      bytes.Compare(result[i].Input.Hash[:], result[j].Input.Hash[:]) < 0
})
```

Problem Scenarios

When two or more UTXOs share the same output slot and input hash (meaning they came from the same transaction within a block), their sorting order is not guaranteed. This violates the requirement for creating identical batches, which can lead to a system halt.

Recommendation

To ensure reliable UTXOs sorting, add a comparison of input indices for cases where output slots and input hashes are identical.

Incorrect quorum formula fails to guarantee BFT safety in edge cases

ID	IF-FINDING-003
Severity	Critical
Impact	3 - High
Exploitability	3 - High
Type	Protocol
Status	Patched Without Reaudit

Involved artifacts

- apex-bridge/common/utils.go

Description

When Cardano batch is generated, Cardano transaction that represents a batch is created using `CreateTx()` function from `cardanotx` package. One of the parameters the mentioned function receives is `cardano.TxInputInfos` (here) structure which contains `MultiSig` and `MultiSigFee` fields of type `cardano.TxInputInfo`. One of the fields in those structures is a policy script field (here). A policy script, since it is a multisig, requires a certain amount of signatures which is decided by the function `common.GetRequiredSignaturesForConsensus()` (here). Function is shown in the code snippet below.

```
func GetRequiredSignaturesForConsensus(cnt uint64) uint64 {
  return (cnt*2 + 2) / 3
}
```

Problem Scenarios

BFT algorithms require more than two thirds of the votes in the system, meaning that it requires `2*validator_count/3 + 1`. However, `GetRequiredSignaturesForConsensus()` and the mentioned formula do not return the same result for certain edge cases.

In a system which has `6` (or `9`, `12`, `15`, etc.) validators, these two functions will return different values. The value returned by `GetRequiredSignaturesForConsensus()` will equal exact `2*validator_count/3` meaning that the quorum could be reached with less than `2*validator_count/3 + 1` votes. This will happen due to rounding, since `uint` division will always round down.

With `6` validators, `(2n + 2)/3 = (12 + 2)/3 = 14/3 = 4.66`, which floors to `4`, will be output of the function, while `2n/3 + 1= 2*6/3 + 1 = 12/3 + 1 = 4 + 1 = 5` would be the accurate value required to keep system resilient to Byzantine faults.

Recommendation

Implement mathematic formula in such a way that it matches the theoretical concepts.

```
func GetRequiredSignaturesForConsensus(cnt uint64) uint64 {
  return (cnt * 2) / 3 + 1
}
```

Centralisation risk due to owner with privileged rights

ID	IF-FINDING-004
Severity	High
Impact	3 - High
Exploitability	2 - Medium
Type	Implementation
Status	Acknowledged

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts

Description

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds. All `initialize()` functions of the contracts take an `_owner` parameter, which is the address of a EOA to which the ownership of the contract is transferred.

Problem Scenarios

Owners have rights to register new chains, defund multisig accounts, set new fund admins, among other admin tasks. Giving such privileges to a single account creates a single point of failure: if the admin key is lost, stolen, or corrupted, the entire system can be compromised—potentially locking or draining funds of users.

Recommendation

Consider distributing the admin access across multiple signatures to eliminate single-point-of-failure risks.

The system heavily depends on external Gouroboros library

ID	IF-FINDING-005
Severity	High
Impact	3 - High
Exploitability	2 - Medium
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- All files in the indexer package
- apex-bridge/batcher/cardano_chain_operations.go

Description

The system relies on the external Gourboros library to establish and maintain Cardano chain node connections. This library provides primitive data types for the Ouroboros protocol and exposes mechanisms for roll-forward and roll-backward operations to synchronize with the latest chain states.

The indexer liveness relies on the correctness of the library, specifically the following assumptions:

- All blocks received have monotonically increasing block numbers.
- The connection is synchronous and does not execute roll-forward and roll-back operations in parallel.
- Chain reorganization detection delays are upper-bounded.
- Blocks integrity is validated.

The batcher's safety also relies on the assumption that it can fetch the protocol parameters in order to create Cardano transactions.

In addition, the indexer currently uses an old Gouroboros version `v0.103.1` thus missing some fixes.

Problem Scenarios

Gouroboros is an early-stage library, making it risky to build production-critical infrastructure on top of it. Any unexpected behavior that violates the assumptions mentioned above could cause the system to halt. Moreover, there's no guarantee that the Gouroboros library will be maintained long-term.

Recommendation

Here is an approach to reduce the dependency risk:

- Isolate the Gouroboros logic in an thin adapter layer by wrapping its logic behind interfaces.
- Never call Gouroboros APIs directly from the system business logic.
- Normalize data by creating your own representation of Blocks, Transactions and Epoch information.
- Consider Ogmios as a more stable backup option—enabling a switch if Gouroboros becomes unmaintained.
- Tracking closely the development status of Gouroboros library.

Quorum may be reached for multiple refund request claims that only differ on one field

ID	IF-FINDING-006
Severity	Medium
Impact	3 - High
Exploitability	1 - Low
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts/Claims.sol

Description

There is no protection in the bridge smart contracts against successfully processing multiple refund request claims that only differ in a single field. The test below shows that two claims reach quorum that only differ on the `refundTransactionHash` field (which is not used at the moment):

```
it("Allows quorum for different RRC with only different refund tx hash", async function () {
  const {
    bridge,
    claimsHelper,
    claims,
    owner,
    validators,
    chain1,
    chain2,
    validatorClaimsRRC,
    validatorsCardanoData
  } = await loadFixture(deployBridgeFixture);

  await bridge.connect(owner).registerChain(chain2, 100, validatorsCardanoData);

  // Create a second claim with different hash but same other data
  const modifiedClaim = JSON.parse(JSON.stringify(validatorClaimsRRC));
  modifiedClaim.refundRequestClaims[0].refundTransactionHash =
    "0x" + "2".repeat(64); // Different hash

  // Group of validators submit original claim
  await bridge.connect(validators[0]).submitClaims(validatorClaimsRRC);
  await bridge.connect(validators[1]).submitClaims(validatorClaimsRRC);
  await bridge.connect(validators[2]).submitClaims(validatorClaimsRRC);

  // Group of validators submit modified claim
  await bridge.connect(validators[0]).submitClaims(modifiedClaim);
  await bridge.connect(validators[1]).submitClaims(modifiedClaim);
  await bridge.connect(validators[2]).submitClaims(modifiedClaim);

  const abiCoder = new ethers.AbiCoder();
  const encodedPrefix = abiCoder.encode(["string"], ["RRC"]);
```

```
// Calculate hash for original claim
const encoded1 = abiCoder.encode(
  ["bytes32", "bytes32", "uint256", "bytes", "string", "uint64", "uint8", "bool"],
  [
    validatorClaimsRRC.refundRequestClaims[0].originTransactionHash,
    validatorClaimsRRC.refundRequestClaims[0].refundTransactionHash,
    validatorClaimsRRC.refundRequestClaims[0].originAmount,
    validatorClaimsRRC.refundRequestClaims[0].outputIndexes,
    validatorClaimsRRC.refundRequestClaims[0].originSenderAddress,
    validatorClaimsRRC.refundRequestClaims[0].retryCounter,
    validatorClaimsRRC.refundRequestClaims[0].originChainId,
    validatorClaimsRRC.refundRequestClaims[0].shouldDecrementHotWallet,
  ]
);

const encoded40_1 =
  "0x0000000000000000000000000000000000000000000000000000" +
  "0000000000000000000000400000000000000000000000000000" +
  "0000000000000000000000000000000000000000000000000080" +
  encodedPrefix.substring(66) +
  encoded1.substring(2);

const hash1 = ethers.keccak256(encoded40_1);

// Calculate hash for modified claim
const encoded2 = abiCoder.encode(
  ["bytes32", "bytes32", "uint256", "bytes", "string", "uint64", "uint8", "bool"],
  [
    modifiedClaim.refundRequestClaims[0].originTransactionHash,
    modifiedClaim.refundRequestClaims[0].refundTransactionHash,
    modifiedClaim.refundRequestClaims[0].originAmount,
    modifiedClaim.refundRequestClaims[0].outputIndexes,
    modifiedClaim.refundRequestClaims[0].originSenderAddress,
    modifiedClaim.refundRequestClaims[0].retryCounter,
    modifiedClaim.refundRequestClaims[0].originChainId,
    modifiedClaim.refundRequestClaims[0].shouldDecrementHotWallet,
  ]
);

const encoded40_2 =
  "0x0000000000000000000000000000000000000000000000000000" +
  "0000000000000000000000400000000000000000000000000000" +
  "0000000000000000000000000000000000000000000000000080" +
  encodedPrefix.substring(66) +
  encoded2.substring(2);

const hash2 = ethers.keccak256(encoded40_2);

// Verify that the hashes are different
expect(hash1).to.not.equal(hash2);

// Verify that neither claim has reached quorum yet
expect(await claimsHelper.numberOfVotes(hash1)).to.equal(3);
expect(await claimsHelper.numberOfVotes(hash2)).to.equal(3);

// Try to reach quorum for first claim
await bridge.connect(validators[3]).submitClaims(validatorClaimsRRC);
```

```
  // First claim should now be confirmed
  expect(await claimsHelper.numberOfVotes(hash1)).to.equal(4);

  // Try to reach quorum for second claim
  await bridge.connect(validators[3]).submitClaims(modifiedClaim);

  // Second claim should now be confirmed
  expect(await claimsHelper.numberOfVotes(hash2)).to.equal(4);
});
```

Problem Scenarios

While this scenario should not occur with an honest, correctly functioning trusted oracle, a bug could cause oracles to accidentally submit multiple refund request claims for the same refund. We consider the exploitability to be low given this assumption. However, since all oracles run the same code, a software bug could potentially trigger this issue. Processing more than one refund request claim would trigger multiple refunds on the source chain—allowing users to recover the same funds multiple times.

Recommendation

Currently there is no correlation in the bridge smart contracts between the refund request claim and the bridging transaction that should revert on the source chain to refund the sender. Implementing a mechanism to keep track for which bridging transactions a refund request claim has already been processed would protect the contracts against replay scenarios.

Out-of-sync batchers are not able to sync with others

ID	IF-FINDING-007
Severity	Medium
Impact	3 - High
Exploitability	1 - Low
Type	Protocol
Status	Acknowledged

Involved artifacts

For Reactor & Skyline:

- batcher/batcher.go

Description

Batcher run an infinite `for` loop which has a timeout for certain time period before starting the batching mechanism. First there is a check (here) to make sure that batcher's indexer's view of the latest block point is larger or equal to the smart contract's view of the last observed block (here, here).

Problem Scenarios

The main issue lies in the retry mechanism's lack of consideration for unsynchronized batchers. If the batcher is not synchronized, a log message will indicate this and note that batch creation was skipped (here). However, the timeout remains unchanged—there is no distinction in delay between successful and failed batch creation attempts. If an indexer is lagging behind, the system assumes it will catch up in the next iteration of the loop (or sometime in the future). This highlights the absence of a mechanism to enforce synchronization between indexers.

Additionally, there may be problems related to data sources. Each indexer connects to a different Cardano node, and it is possible that an indexer connects to a node that consistently provides stale data, rendering it ineffective in quorum participation.

Finally, in this system, the limited number of validators introduces another concern: if a validator does not participate in the quorum, it could lead to potential liveness issues.

Recommendation

We recommend implementing some mechanism that would force validators to sync their indexers if they are out of sync for a certain period of time.

Lack of validation for validators' verifying key when owner registers a chain

ID	IF-FINDING-008
Severity	Medium
Impact	3 - High
Exploitability	1 - Low
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts/Validators.sol
- apex-bridge-smartcontracts/contracts/Bridge.sol

Description

When the owner of the `Bridge` contract registers a new chain using function `registerChain()` an array with the verifying keys of each bridge validator for the chain is passed. The data is stored during the execution of function `setValidatorsChainData()` in the `chainData` mapping. Unlike in function `registerChainGovernance()`, where the verifying key in the parameter `_validatorChainData` is used to verify the signature of a message and therefore basic sanity checks are performed to guarantee that the key is sound, in function `registerChain()` or `setValidatorChainData()` there are no checks performed on the key.

Problem Scenarios

Without any validation, the bridge would end up in an invalid state and errors would only surface afterwards, when verifying signatures when a signed batch is submitted. Even though the function `registerChain()` is gated and only the owner can execute it, the probability of error is not negligible.

Recommendation

Add sanity checks in either function `registerChain()` or function `setValidatorChainData()` to make sure that the verifying keys are valid.

Quorum may be reached for multiple batch executed claims or multiple batch execution failed claims (or both) for the same batch

ID	IF-FINDING-009
Severity	Medium
Impact	3 - High
Exploitability	1 - Low
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts/Claims.sol

Description

There is no protection in the bridge smart contracts against:

- Successfully processing multiple different batch executed claims for the same batch.
- Successfully processing multiple different batch execution failed claims for the same batch.
- Successfully processing a batch executed claim and a batch execution failed claim for the same batch.

Even though this might be a design choice given the assumption that oracles will behave correctly, it is possible, for any given batch, to submit and reach quorum in the above mentioned situations. For example, reaching quorum for both a batch executed claim and a batch execution failed claim is possible, as the following test shows:

```
it("Allows quorum for both BEC and BEFC with same batch ID", async function () {
  const { bridge, claimsHelper, owner, validators, chain2, validatorClaimsBEC,
  ↪  validatorClaimsBEFC, validatorsCardanoData } =
  await loadFixture(deployBridgeFixture);

  // Register the chain
  await bridge.connect(owner).registerChain(chain2, 100, validatorsCardanoData);

  // Create our claims with same batch ID but different purposes
  const batchId = validatorClaimsBEC.batchExecutedClaims[0].batchNonceId;
  validatorClaimsBEFC.batchExecutionFailedClaims[0].batchNonceId = batchId;

  // Group of validators submit original claim
  await bridge.connect(validators[0]).submitClaims(validatorClaimsBEC);
  await bridge.connect(validators[1]).submitClaims(validatorClaimsBEC);
  await bridge.connect(validators[2]).submitClaims(validatorClaimsBEC);

  // Group of validators submit modified claim
  await bridge.connect(validators[0]).submitClaims(validatorClaimsBEFC);
  await bridge.connect(validators[1]).submitClaims(validatorClaimsBEFC);
  await bridge.connect(validators[2]).submitClaims(validatorClaimsBEFC);

  // Calculate BEC hash
  const abiCoder = new ethers.AbiCoder();
  const encodedPrefixBEC = abiCoder.encode(["string"], ["BEC"]);
  const encodedBEC = abiCoder.encode(
    ["bytes32", "uint64", "uint8"],
    [
```

```
      validatorClaimsBEC.batchExecutedClaims[0].observedTransactionHash,
      validatorClaimsBEC.batchExecutedClaims[0].batchNonceId,
      validatorClaimsBEC.batchExecutedClaims[0].chainId,
    ]
  );

  const encoded40BEC =
    "0x0000000000000000000000000000000000000000" +
    "0000000000000000000000000000000000000080" +
    encodedBEC.substring(2) +
    encodedPrefixBEC.substring(66);

  const hashBEC = ethers.keccak256(encoded40BEC);

  // Calculate BEFC hash
  const encodedPrefixBEFC = abiCoder.encode(["string"], ["BEFC"]);
  const encodedBEFC = abiCoder.encode(
    ["bytes32", "uint64", "uint8"],
    [
      validatorClaimsBEFC.batchExecutionFailedClaims[0].observedTransactionHash,
      validatorClaimsBEFC.batchExecutionFailedClaims[0].batchNonceId,
      validatorClaimsBEFC.batchExecutionFailedClaims[0].chainId,
    ]
  );

  const encoded40BEFC =
    "0x0000000000000000000000000000000000000000" +
    "0000000000000000000000000000000000000080" +
    encodedBEFC.substring(2) +
    encodedPrefixBEFC.substring(66);

  const hashBEFC = ethers.keccak256(encoded40BEFC);

  // Verify that the hashes are different
  expect(hashBEC).to.not.equal(hashBEFC);

  // Verify that neither claim has reached quorum yet
  expect(await claimsHelper.numberOfVotes(hashBEC)).to.equal(3);
  expect(await claimsHelper.numberOfVotes(hashBEFC)).to.equal(3);

  // Try to reach quorum for first claim
  await bridge.connect(validators[3]).submitClaims(validatorClaimsBEC);

  // First claim should now be confirmed
  expect(await claimsHelper.numberOfVotes(hashBEC)).to.equal(4);

  // Try to reach quorum for second claim
  await bridge.connect(validators[3]).submitClaims(validatorClaimsBEFC);

  // First claim should now be confirmed
  expect(await claimsHelper.numberOfVotes(hashBEFC)).to.equal(4);

  // Verify that both claims reference same batch
  expect(validatorClaimsBEC.batchExecutedClaims[0].batchNonceId)
    .to.equal(validatorClaimsBEFC.batchExecutionFailedClaims[0].batchNonceId);
});
```

Problem Scenarios

While these scenarios should not occur with an honest, correctly functioning trusted oracle, a bug could cause oracles to accidentally submit multiple batch executed claims, or multiple batch execution failed claims, or both a batch executed claim and a batch execution failed claim, for the same batch in all cases. We consider the exploitability to be low given this assumption. However, since all oracles run the same code, a software bug could potentially trigger this issue. Processing more than one batch execution failed claims would trigger multiple refunds on the source chain—allowing users to recover the same funds multiple times.

Another problematic scenario arises when a batch executed claim or a batch execution failed claim is processed for a past batch. When the claim reaches quorum, the `lastBatchTxNonceId` for the given chain ID is updated (here and here for batch executed claims and batch execution failed claims, respectively) with the nonce of the last transaction in the batch. This may set the value backwards, and when generating the next batch, the `getConfirmedTransactions` query would return transactions for a batch that may have already been processed.

Recommendation

One possible way to prevent this to happen is to delete the confirmed signed batch from the `confirmedSignedBatches` mapping. By deleting the confirmed signed batch when a batch executed claim or a batch execution failed claim is processed, then further claims for the same batch would fail. For example, in the implementation of `_submitClaimsBEC()` the changes, after reaching quorum, could be something like the following:

```
if (_quorumReached) {
  // current batch block must be reset in any case because otherwise bridge will be blocked
  claimsHelper.resetCurrentBatchBlock(chainId);

  ConfirmedSignedBatchData memory _confirmedSignedBatch =
↪   claimsHelper.getConfirmedSignedBatchData(
    chainId,
    batchId
  );

  if (_confirmedSignedBatch.firstTxNonceId == 0 &&
      _confirmedSignedBatch.lastTxNonceId == 0 &&
      _confirmedSignedBatch.isConsolidation == false) {
    revert("Confirmed signed batch not found");
  }

  // do not process included transactions or modify batch creation state if it is a
  ↪  consolidation
  if (_confirmedSignedBatch.isConsolidation) {
    return;
  }

  lastBatchedTxNonce[chainId] = _confirmedSignedBatch.lastTxNonceId
  nextTimeoutBlock[chainId] = block.number + timeoutBlocksNumber;

  claimsHelper.deleteConfirmedSignedBatch(chainId, batchId);
}
```

where `deleteConfirmedSignedBatch` would be implemented in `ClaimsHelper` contract as:

```
function deleteConfirmedSignedBatch(uint8 chainId, uint64 batchId) external
↪   onlySignedBatchesOrClaims {
  confirmedSignedBatches[chainId][batchId] = ConfirmedSignedBatchData(0, 0, false);
}
```

An additional benefit of deleting the value for the chain ID and batch ID combination is that it triggers a gas refund

during the transaction.

Similar changes could be implemented in `_submitClaimsBEFC()`.

Addresses expected to be contract accounts may be set to EOA addresses

ID	IF-FINDING-010
Severity	Medium
Impact	3 - High
Exploitability	1 - Low
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts/Validators.sol
- apex-bridge-smartcontracts/contracts/Slots.sol
- apex-bridge-smartcontracts/contracts/Admin.sol
- apex-bridge-smartcontracts/contracts/Bridge.sol
- apex-bridge-smartcontracts/contracts/SignedBatches.sol
- apex-bridge-smartcontracts/contracts/Claims.sol
- apex-bridge-smartcontracts/contracts/ClaimsHelper.sol

Description

In the `setDependencies()` function of the contracts a number of addresses are taken as inputs. These addresses are expected to be for contract accounts of the dependent contracts. But currently there is no validation to make sure that Externally Owned Account (EOA) addresses are passed in. The following test shows how the `setDependencies()` functions of `Slots` contract can be executed with two EOA addresses for the `_bridgeAddress` and `_validatorsAddress` parameters:

```
it("Should allow EOA addresses in setDependencies but fail when using them", async function ()
↪  {
  const [owner, eoa1, eoa2, validator] = await ethers.getSigners();

  // Deploy Slots contract
  const Slots = await ethers.getContractFactory("Slots");
  const slotsLogic = await Slots.deploy();

  const SlotsProxy = await ethers.getContractFactory("ERC1967Proxy");
  const initData = Slots.interface.encodeFunctionData("initialize", [owner.address,
  ↪  owner.address]);

  const slotsProxy = await SlotsProxy.deploy(
    await slotsLogic.getAddress(),
    initData
  );

  const slots = Slots.attach(slotsProxy.target);

  // Set EOA addresses as dependencies - this should succeed
  await expect(
    slots.setDependencies(eoa1.address, eoa2.address)
  ).not.to.be.reverted;

  // Create some test block data
  const blocks = [{
```

```
    blockSlot: 1,
    blockHash: "0x7465737400000000000000000000000000000000000000000000000000000000"
  }];

  // Try to use the contract - this should fail because
  // eoa2 can't act as a Validators contract
  await expect(
    slots.connect(eoa1).updateBlocks(1, blocks, validator)
  ).to.be.revertedWithoutReason();
});
```

Problem Scenarios

As the test above shows, the `setDependencies()` function successfully executes (in the particular case of the `Slots` contract, the type casting `Validators(_validatorsAddress)` succeeds because it doesn't perform any runtime checks), but if accidentally EOA addresses have been passed as parameters, the issue might only become apparent later on:

- when another function is called with a modifier that checks one of the addresses,
- or when a function calls another function on a dependent contract (the function would revert because when it tries to call the dependent contract's function, the call will fail since EOAs has no code to execute).

Recommendation

Implementing an `isContract()` function that checks if the input address corresponds to a contract account:

```
function isContract(address addr) internal view returns (bool) {
  uint256 size;
  assembly { size := extcodesize(addr) }
  return size > 0;
}
```

Then this function could be called in `setDependencies`. For example, for `Slots` contract:

```
function setDependencies(address _bridgeAddress, address _validatorsAddress) external onlyOwner
↪  {
  require(isContract(_bridgeAddress), "Address must be a contract");
  require(isContract(_validatorsAddress), "Address must be a contract");

  bridgeAddress = _bridgeAddress;
  validators = Validators(_validatorsAddress);
}
```

This checks prevent the issue by failing fast when trying to set invalid addresses, rather than allowing the contract to enter an invalid state that will fail later. And it also provides a clear error message.

Contracts may be initialised with zero addresses for owner and upgrade admin

ID	IF-FINDING-011
Severity	Medium
Impact	3 - High
Exploitability	1 - Low
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts/Validators.sol
- apex-bridge-smartcontracts/contracts/Slots.sol
- apex-bridge-smartcontracts/contracts/Admin.sol
- apex-bridge-smartcontracts/contracts/Bridge.sol
- apex-bridge-smartcontracts/contracts/SignedBatches.sol
- apex-bridge-smartcontracts/contracts/Claims.sol
- apex-bridge-smartcontracts/contracts/ClaimsHelper.sol

Description

The `initialize()` functions of the contracts accept two address parameters: `_owner` and `upgradeAdmin`. They are the addresses for the accounts that will exercise administrative control over the contracts. Currently the functions lack validation to make sure that the addresses cannot be the zero address, as the test below shows for the `initialize()` function of `Validators` contract:

```
it("Initializes with zero addresses for owner and upgrade admin", async function () {
  const [, validator1, validator2] = await ethers.getSigners();

  // Deploy implementation contract
  const Validators = await ethers.getContractFactory("Validators");
  const validatorsLogic = await Validators.deploy();

  // Deploy proxy contract
  const ValidatorsProxy = await ethers.getContractFactory("ERC1967Proxy");

  // Create validator addresses array
  const validatorAddresses = [
    validator1.address,
    validator2.address
  ];

  const ZERO_ADDRESS = "0x0000000000000000000000000000000000000000";

  // Prepare initialization data with zero addresses
  const initData = Validators.interface.encodeFunctionData("initialize", [
    ZERO_ADDRESS, // owner address
    ZERO_ADDRESS, // upgrade admin address
    validatorAddresses
  ]);

  // Deploy proxy with initialization
```

```
  const validatorsProxy = await ValidatorsProxy.deploy(
    await validatorsLogic.getAddress(),
    initData
  );

  // Get Validators interface for the proxy address
  const validators = Validators.attach(validatorsProxy.target);

  // Verify the contract state
  const actualOwner = await validators.owner();
  expect(actualOwner).to.equal(ZERO_ADDRESS);

  // Try to call an onlyOwner function
  const randomAddress = ethers.Wallet.createRandom().address;
  await expect(
    validators.setDependencies(randomAddress)
  ).to.be.revertedWith("Ownable: caller is not the owner");
});
```

Problem Scenarios

If the owner or upgrade admin of the contracts is (accidentally) set to a zero address, then the contracts would be misconfigured and possibly rendered unusable.

Recommendation

Add a check on the `initialize()` functions to make sure that calls to the function with the parameters `_owner` and `upgradeAdmin` set to the zero address are rejected. For example:

```
function initialize(address _owner, address _upgradeAdmin, address[] calldata _validators)
↪  public initializer {
  require(_owner != address(0), "Owner cannot be zero address");
  require(_upgradeAdmin != address(0), "Upgrade admin cannot be zero address");

  _transferOwnership(_owner);
  upgradeAdmin = _upgradeAdmin;
  for (uint8 i; i < _validators.length; i++) {
    addressValidatorIndex[_validators[i]] = i + 1;
  }
  validatorsCount = uint8(_validators.length);
}
```

If needed, there are also more gas efficient ways to check for zero address.

Batcher heavy relies on out of scope packages

ID	IF-FINDING-012
Severity	Medium
Impact	3 - High
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

For Reactor & Skyline:

- apex-bridge/batcher/cardano_chain_operations.go
- batcher/evm_chain_operations.go

Description

The bridge implementation relies heavily on external packages developed by the same team (but outside the scope of this audit) for critical operations. Specifically, packages like cardano-infrastructure/wallet are used for essential bridge functionality, including transaction creation and batch signing management. Also, EVM batcher relies on EVM blockchain-event-tracker/store as EVM indexer. These dependencies could represent a significant security risk as they handle critical operations but have not undergone the same level of security scrutiny as the certain parts of the core bridge code.

Problem Scenarios

Based on the description, we have identified several problem scenarios:

- **Vulnerability propagation**: Any vulnerability in the unaudited external packages directly affects the security of the bridge. For example, if the transaction creation functionality in cardano-infrastructure/wallet contains a flaw, it could lead to transaction manipulation, incorrect signature validation, or resource exhaustion attacks against the bridge.
- **Inconsistent security standards**: The external packages may not adhere to the same security practices and standards as the audited codebase, creating inconsistencies in the overall security posture.
- **Scope limitations**: Important security mechanisms implemented in these external packages cannot be fully evaluated during this audit, potentially masking critical issues.
- **Maintenance risk**: Updates to these external dependencies might introduce new vulnerabilities or break existing security assumptions without triggering a new audit.
- **Transitive dependencies**: These external packages may themselves depend on other unaudited packages, creating a chain of trust issues that extend beyond direct dependencies.

Recommendation

Some of our recommendations would be:

- **Audit**: Have critical external packages audited, particularly those developed in-house that handle security-sensitive operations like transaction creation, signature validation, and wallet management.
- **Comprehensive testing**: Implement extensive testing specifically for the integration points with external packages, including boundary cases and failure scenarios.
- **Version control**: Strictly lock external package versions and implement a formal review process for any version updates.
- **Security documentation**: Maintain detailed documentation of the security assumptions and guarantees provided by external packages, along with any known limitations.

Missed opportunity to consolidate UTXOs

ID	IF-FINDING-013
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

- batcher/cardano_chain_operations.go

Description

`getNeededUtxos()` returns needed input UTXOs from the `inputUTXOs` parameter, from first to last until `desiredAmount` has been met or `maxUtxoCount` reached. Condition to consider UTXOs successfully selected is that either that `chosenUTXOsSum` is equal to `desiredAmount` or larger than `desiredAmount + minUtxoAmount` (here) where `minUtxoAmount` represents the minimum amount that UTXO can have in order for the system to be able to create it. If the amount conditions were not satisfied, then the system should trigger consolidation or return error that it couldn't select UTXOs (here).

Problem Scenarios

The problematic scenario lies in the edge case. `totalUTXOsSum` variable tracks total sum of available UTXOs. If `totalUTXOsSum` is not larger than `minUtxoAmount + desiredAmount` then the system would not trigger consolidation. This will lead to batcher not being able to create a batch and not triggering consolidation. However, in edge case `totalUTXOsSum = desiredAmount`, the system should be able to create a batch after consolidation, however that opportunity will be missed, rendering that batch unable to be generated.

Recommendation

We recommend extending the condition in such a way that would force consolidation if deemed that batch creation would succeed after consolidation.

```
func getNeededUtxos(
  inputUTXOs []*indexer.TxInputOutput,
  desiredAmount uint64,
  minUtxoAmount uint64,
  utxoCount int,
  maxUtxoCount int,
  takeAtLeastUtxoCount int,
) (chosenUTXOs []*indexer.TxInputOutput, err error) {
  txCostWithMinChange := minUtxoAmount + desiredAmount
  chosenUTXOsSum := uint64(0)
  totalUTXOsSum := uint64(0)
  isUtxosOk := false

  ...

  if !isUtxosOk {
    if totalUTXOsSum >= txCostWithMinChange ||
       totalUTXOsSum == desiredAmount {
      return nil, fmt.Errorf("%w: %d vs %d", errUTXOsLimitReached,
```

```
                                totalUTXOsSum, txCostWithMinChange)
    }

    return nil, fmt.Errorf("%w: %d vs %d", errUTXOsCouldNotSelect,
                               totalUTXOsSum, txCostWithMinChange)
  }

  return chosenUTXOs, nil
}
```

Loops over unbounded arrays

ID	IF-FINDING-014
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts/Claims.sol
- apex-bridge-smartcontracts/contracts/Slots.sol

Description

When processing the validator claims in `submitClaims()` function there is a loop for each type of claim where the code goes through the claims in the array of each claim type:

```
for (uint i; i < bridgingRequestClaimsLength; i++)
...
for (uint i; i < batchExecutedClaimsLength; i++)
...
for (uint i; i < batchExecutionFailedClaimsLength; i++)
...
for (uint i; i < refundRequestClaimsLength; i++)
...
for (uint i; i < hotWalletIncrementClaimsLength; i++)
```

There is currently no validation to check the lengths of the arrays in `ValidatorClaims`, which means that the arrays could potentially have an unbounded length.

Similar issue exists in the function `updateBlocks()` where no validation for input `blocks` of type `CardanoBlock[]` is implemented to guarantee that the length is limited to an reasonable, expected maximum value; and also in function `_setConfirmedTransactions()` where a loop goes through the array of receivers.

With lower probability if the care is taken when setting and updating the value of `maxNumberOfTransactions`, this problem could also manifest in function `_submitClaimsBEFC()` where a `for` statement exists that loops through the confirmed transactions between two nonces:

```
for (uint64 i = _firstTxNonce; i <= _lastTxNonce; i++) {
  ...
}
```

Problem Scenarios

Even though the situation of looping through unbounded arrays should be unlikely assuming that trusted oracles act honestly and function correctly, and `maxNumberOfTransactions` is set to a reasonable value, a bug in the oracle software, or lack of business logic when batching claims or blocks, or accidentally setting `maxNumberOfTransactions` to a large value, could all be factors that could result in loops that process too many elements. If this happens, processing all elements may exceed the block gas limit, causing transactions to fail. This not only creates unpredictable gas costs but also makes the functions vulnerable to liveness issues or even denial-of-service (DoS) attacks.

Recommendation

Similarly as implemented in PR#94, add validation that checks that arrays have a limited length.

Duplicate validator addresses may inflate quorum requirement

ID	IF-FINDING-015
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts/Validators.sol

Description

The `initialize()` function in `Validators.sol` takes as parameter the `address` array `_validators` with the addresses for the validator set of the bridge and connected blochains. The function does not check if the array contains duplicate values, so it is possible to call the function with an array that does not contain unique values, as the test below confirms:

```
it("Accepts duplicate validator addresses in initialize function", async function () {
  const [owner, validator1, validator2] = await ethers.getSigners();

  // Deploy implementation contract
  const Validators = await ethers.getContractFactory("Validators");
  const validatorsLogic = await Validators.deploy();

  // Deploy proxy contract
  const ValidatorsProxy = await ethers.getContractFactory("ERC1967Proxy");

  // Create array with duplicate addresses
  const validatorAddresses = [
    validator1.address,
    validator2.address,
    validator1.address // Duplicate address
  ];

  // Prepare initialization data
  const initData = Validators.interface.encodeFunctionData("initialize", [
    owner.address,
    owner.address,
    validatorAddresses
  ]);

  // Deploy proxy with initialization
  const validatorsProxy = await ValidatorsProxy.deploy(
    await validatorsLogic.getAddress(),
    initData
  );

  // Get Validators interface for the proxy address
  const validators = Validators.attach(validatorsProxy.target);
```

```
  // Verify the contract state
  const validatorsCount = await validators.validatorsCount();
  expect(validatorsCount).to.equal(3); // Total number of entries in the array

  // Check index for validator1 (should be 3 due to last occurrence)
  const index1 = await validators.getValidatorIndex(validator1.address);
  expect(index1).to.equal(3); // Should be 3 (last occurrence)

  // Check index for validator2
  const index2 = await validators.getValidatorIndex(validator2.address);
  expect(index2).to.equal(2); // Should be 2 (its only occurrence)

  // Verify validator status
  expect(await validators.isValidator(validator1.address)).to.be.true;
  expect(await validators.isValidator(validator2.address)).to.be.true;
});
```

The test shows that the `validatorsCount` field holds a value of 3, while actually there are only two elements in the `addresValidatorIndex` mapping.

Problem Scenarios

The function `getQuorumNumberOfValidators()` would return an inflated count (because `validatorsCount` value is higher than the actual number of unique validators), which elevates the number needed of validators to sign claims and batches for reaching quorum, potentially impacting the performance and liveness of the system.

Recommendation

Add validation to `initialize()` function, so that if a validator address has already been added to the `addressValidatorIndex` mapping, then the function will throw an exception and terminate:

```
function initialize(address _owner, address _upgradeAdmin, address[] calldata _validators)
↪  public initializer {
  _transferOwnership(_owner);
  upgradeAdmin = _upgradeAdmin;
  for (uint8 i; i < _validators.length; i++) {
    require(addressValidatorIndex[_validators[i]] != 0, "validator already exists");
    addressValidatorIndex[_validators[i]] = i + 1;
  }
  validatorsCount = uint8(_validators.length);
}
```

Redundancy in GenerateBatchTransaction code

ID	IF-FINDING-016
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge/batcher/cardano_chain_operations.go

Description

During `GenerateBatchTransaction()` function execution, `generateBatchTransaction()` function is called first with intention to generate a batch, and if it returns either `errUTXOsLimitReached` or `errTxSizeTooBig` it will call `generateConsolidationTransaction()` intending to generate a consolidation batch. However, there is a lot of redundant operations that are generating unchanged data in such a short period of time (here and here), such as validator keys, policy scripts, multisig addresses and protocol parameters.

Problem Scenarios

These operations cost time and resources. For example, `getCardanoData()` queries bridge smart contracts for validator keys and validates them, `CreateBatchMetaData()` just serializes a structure with `batchNonceID` into bytes, `GetPolicyScripts()` creates policy scripts using `validator_data`, `GetMultisigAddresses()` generates addresses for those policy scripts, while `GetProtocolParameters()` opens a connection to Gouroboros protocol and gets `protocol_params`. Opening a connection towards Gouroboros protocol could introduce timeouts and potential failures, while protocol parameters are not intended to be changed as often.

Executing set of this functions twice is not optimal, and not doing so can save time and resources, increasing the performance.

Recommendation

Extract those functions outside of `generateBatchTransaction()` and `generateConsolidationTransaction()` and pass those values into previously mentioned functions.

Validators have different address configurations in their block indexer

ID	IF-FINDING-017
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- indexer/block_indexer.go

Description

Validators have set `AddressCheck` and `AddressOfInterest` as part of configurations.

Problem Scenarios

If one or more validators have mismatched address configurations in their indexer as the result they could filter and store different UTXO and transaction sets, preventing the validators from reaching quorum on bridge claims and batches, and halting the bridge.

Recommendation

Apex Dev team must ensure that all validators run their validator with identical address configuration on their block indexers.

Cardano transactions with zero amount will be created when there is no multisig and fee UTXOs

ID	IF-FINDING-018
Severity	Informational
Impact	1 - Low
Exploitability	0 - None
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- batcher/cardano_chain_operations.go

Description

During batch generation, the system may decide to create a consolidation batch under certain circumstances. In that case, the system gets UTXOs from the multisig address on the destination chain and UTXOs from the multisigFee adddress on the destination chain and checks if there are any fees left to use after filtering out tokens(here).

Problem Scenarios

Our analysis has found that the system may attempt to create transactions with empty or invalid output sets under certain conditions. Specifically, in the `generateConsolidationTransaction()` function, if `multisigUtxos` is empty, the code will still proceed to create a transaction, potentially leading to an invalid state. When examining `CreateTx()`, it appears the function will not explicitly reject an empty outputs array. Instead, it will attempt to create a transaction, adding at least a `fee` output even when no other outputs exist. This can lead to transactions that serve no consolidation purpose while still consuming fees, which could potentially fail on the destination.

Recommendation

The code should have checks to ensure that created batches will actually serve their purpose. We recommend mplementing a check which will validate that `multisigUtxos` is not empty.

Skyline Batcher miscellaneous code concerns

ID	IF-FINDING-019
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

For Skyline:

- batcher/core/config.go
- batcher/batcher_manager/batcher_manager.go

Description

- `BatcherManagerConfiguration` structure has field `RunMode` (here) which is passed to `getCardanoOperations()` (here). However it is not used anywhere.

Token type not specified in NotEnoughFunds event

ID	IF-FINDING-020
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Skyline:

- apex-bridge-smart-contracts/contracts/Claims.sol

Description

When the either the destination amount for native or wrapped tokens is greater than the available amount on the destination multisig wallet, an event is emitted and the transaction ends transaction (same in `_submitClaimsRRC()` function).

Problem Scenarios

The `NotEnoughFunds` event includes the available amount of funds on the destination multisig wallet, but the event does not include any indication of the denomination of the funds. Therefore, the event could be emitted with either the amount available of native or wrapped tokens. Without indication of the type of token, the event might lack information to clearly indicate what amount in the bridging request claim is causing the issue.

Recommendation

Consider adding extra information to the `claimType` string indicating the type of token or add an extra field to the event. For example:

```
if (chainTokenQuantity[_destinationChainId] < _nativeCurrencyAmountDestination) {
  emit NotEnoughFunds("BRC - Native Token", i, chainTokenQuantity[_destinationChainId]);
  return;
}

if (chainWrappedTokenQuantity[_destinationChainId] < _wrappedTokenAmountDestination) {
  emit NotEnoughFunds("BRC - Wrapped Token", i,
↪  chainWrappedTokenQuantity[_destinationChainId]);
  return;
}
```

Batchers could appear synced if smart contract has outdated block data

ID	IF-FINDING-021
Severity	Informational
Impact	3 - High
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

For Reactor & Skyline:

- batcher/batcher.go
- batcher/cardano_chain_operations.go
- batcher/evm_chain_operations.go

Description

When batcher process starts there is a check (here) which will check if the batcher's indexer has block slot (or block number) data which is newer than the one on the smart contract. Bridge smart contracts rely on the oracles to provide them with observed blocks. The mentioned check is in place to prevent a batcher whose indexer is lagging behind the bridge smart contract to even participate in the quorum.

Problem Scenarios

The problematic scenario is when smart contract has not updated it's observed block for a long period of time. This could indicate serious issues, but those will not be detected on the batcher level. Batcher will assume it is synchronized and will continue submitting batches.

Recommendation

We recommend having mechanism in place which would indicate to the batcher that something is wrong and that it should not sign any more batches.

Bridge smart contracts miscellaneous code improvements

ID	IF-FINDING-022
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts/IBridgeStructs.sol
- apex-bridge-smartcontracts/contracts/Admin.sol
- apex-bridge-smartcontracts/contracts/Bridge.sol
- apex-bridge-smartcontracts/contracts/Validators.sol
- apex-bridge-smartcontracts/contracts/Claims.sol
- apex-bridge-smartcontracts/contracts/ClaimsHelper.sol
- apex-bridge-smartcontracts/contracts/Slots.sol
- apex-bridge-smartcontracts/contracts/SignedBatches.sol
- apex-bridge-smartcontracts/package.json

Recommendation

- The constants prefixed with `PRECOMPILE_` are only used in the `Validators` contract and the contract is not inherited by any other contract, therefore consider adding the `private` visibility modifier.
- The word `nonce` is misspelled in the variable here.
- The steps to check whether a bridging request claim hash has already been voted on and to increment its vote count could be combined by calling function `setVotedOnlyIfNeeded()`, thereby eliminating the need for the separate `isVoteRestrictedfunction`.

```
function _submitClaimsBRC(
  BridgingRequestClaim calldata _claim,
  uint256 i,
  address _caller
) internal {
  bytes32 _claimHash = keccak256(abi.encode("BRC", _claim));
  uint256 _receiversSum = _claim.totalAmount;
  uint8 _destinationChainId = _claim.destinationChainId;

  if (chainTokenQuantity[_destinationChainId] < _receiversSum) {
    emit NotEnoughFunds("BRC", i, chainTokenQuantity[_destinationChainId]);
    return;
  }

  bool _quorumReached = claimsHelper.setVotedOnlyIfNeeded(
    _caller,
    _claimHash,
    validators.getQuorumNumberOfValidators()
  );

  if (_quorumReached) {
    chainTokenQuantity[_destinationChainId] -= _receiversSum;
```

```
    if (_claim.retryCounter == 0) {
    chainTokenQuantity[_claim.sourceChainId] += _receiversSum;
    }

    uint256 _confirmedTxCount = getBatchingTxsCount(_destinationChainId);

    _setConfirmedTransactions(_claim, 0);

   _updateNextTimeoutBlockIfNeeded(_destinationChainId, _confirmedTxCount);
  }
}
```

then `setVotedOnlyIfNeeded()` can be implemented as:

```
function setVotedOnlyIfNeeded(
  address _voter,
  bytes32 _hash,
  uint256 _quorumCnt
) external onlySignedBatchesOrClaims returns (bool) {
  uint8 votes = numberOfVotes[_hash];
  if (votes >= _quorumCnt || hasVoted[_hash][_voter]) {
    return false;
  }

  hasVoted[_hash][_voter] = true;
  numberOfVotes[_hash] = ++votes;
  return votes >= _quorumCnt;
}
```

And function `isVoteRestricted()` could be removed.

- Some of the chain information that is added after registering a new chain doesn't seem to be used (`addressMultisig`, `addressFeePayer`). If these fields are not planned to be used, consider removing them.
- The comments here and here are misleading because this should be hash of the transaction where the batch was executed (or failed) on the destination chain.
- Version 4.9.3 of OpenZeppelin contracts and contracts-upgradable is almost two years old. Consider updating to the latest version (v5.3.0 at the time of writing).
- Most of the Solidity files uses version 0.8.24, but `Admin.sol` and `Bridge.sol` use version 0.8.23. Consider using the same version in all of them (0.8.24 or later).
- The following errors are declared but not used:
  - `AlreadyConfirmed`
  - `NotSignedBatchesOrBridge`
  - `NotEnoughBridgingTokensAvailable`
  - `WrongBatchNonce`

  Consider removing them, if they are not needed.
- The following imports are not used. Consider removing them:
  - `"@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol"` (here),
  - `"@openzeppelin/contracts/utils/Strings.sol"` (here)

Contract upgradability

- All the bridge smart contracts implement the UUPS (Universal Upgradeable Proxy Standard) pattern by inheriting OpenZeppelin's `UUPSUpgradeable` contract. The contracts also inherit `OwnableUpgradeable`

contract, which is usually inherited together with `UUPSUpgradeable` to control who can upgrade the contract. However, the `initialize()` function of the contracts lacks the calls to `__UUPSUpgradeable_init()` and `__Ownable_init()`. Even though the `initialize()` function includes a call to `_transferOwnership()` to move ownership of the contract to the provided address, and potentially the call to `__Ownable_init()` could be skipped, it is the recommended practice to call both `__UUPSUpgradeable_init()` and `__Ownable_init()` for a number of reasons:

  - It signals to readers and tools that this contract supports UUPS upgrades.
  - It future-proofs your contracts, should OpenZeppelin introduce or change any setup logic in later version.
  - Since the constructor of the contracts is disabled, calling `__UUPSUpgradeable_init()` and `__Ownable_init()` will correctly initialise the parent contracts, ensuring that the upgrade mechanism is not broken.

- Consider implementing a pure `version()` function that returns the version of the contracts. For example:

```
function version() public pure returns (uint256) {
  return 1;
}
```

  That future versions of the contract can also implement by returning higher version number, so that it is easy to identify what version of the contract is deployed. Additionally, you could consider checking the version of the new contract in the `_authorizeUpgrade()` function, to make sure that only upgrades to newer versions are allowed.

- Consider using gap variables for future storage expansion, so that it is safe (i.e., preventing storage collision issues) to add new contract state variables in future versions of the contracts. Adding gap variables also future-proofs your contracts, in case they are inherited in the future. For example, at the end of the contract members declaration add:

```
// Reserved storage slots for future variables
uint256[50] private __gap;
```

- There are no tests covering the contract upgrade process, so we recommend adding tests to ensure upgrades are working as expected.

Optimization in validator data key verification

ID	IF-FINDING-023
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge/batcher/cardano_chain_operations.go

Description

After validators data is retrieved from the smart contract. That data has to include verification keys of the validator who is running this particular batcher. Presence of the keys is done with this code snippet (here).

```
for _, validator := range validatorsData {
  hasVerificationKey = hasVerificationKey || bytes.Equal(cco.wallet.MultiSig.VerificationKey,
    cardanowallet.PadKeyToSize(validator.Key[0].Bytes()))

  hasFeeVerificationKey = hasFeeVerificationKey ||
↪   bytes.Equal(cco.wallet.MultiSigFee.VerificationKey,
    cardanowallet.PadKeyToSize(validator.Key[1].Bytes()))
}
```

Problem Scenarios

After the validator has been verified that it is in the `validatorsData` array, the loop will continue until it finishes.

Recommendation

Our recommendation is breaking the loop after the presence of the validator in the validator data has been confirmed.

LevelDB indexer implementation is not tested

ID	IF-FINDING-024
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- cardano-infrastructure/indexer/db/leveldb/leveldb.go
- cardano-infrastructure/indexer/db/leveldb/leveldb_txwriter.go

Description

The LevelDB implementation of the indexer data storage isn't covered by the test.

Recommendation

The LevelDB implementation should be tested, or validator operators should be clearly informed not to use it in production.

Redundant database queries in the processing of confirmed blocks can be optimized

ID	IF-FINDING-025
Severity	Informational
Impact	1 - Low
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

For Reactor & Skyline:

- cardano-infrastructure/indexer/block_indexer.go

Description

Under the default configuration of the block indexer (`config.AddressCheck = 3` and `config.KeepAllTxOutputsInDB = false`), the `processConfirmedBlock` function (here) performs redundant database lookups that can be optimized.

For each transaction, the `GetTxOutput` function is called up to **twice** during the following steps:

- **In `filterTxsOfInterest`** (here): If a transaction does not produce any output to an address of interest, its inputs are checked individually against the database to determine whether they spend a UTXO associated with an address of interest.
- **In `createTx`** (here): Every transaction of interest are processed again, and all its inputs are queried from the database to reconstruct the `TxInputOutput` structure.

As a result, if a transaction is selected due to one of its inputs during `filterTxsOfInterest`, those same inputs will be queried again in `createTx`. Additionally, all transaction inputs of interest are passed to `RemoveTxOutputs`(here) regardless of whether an output exists in the database, resulting in unnecessary database lookups.

Recommendation

To reduce redundant queries, `filterTxsOfInterest` should return an additional `map[Hash][]TxInputOutput` mapping, associating the transaction inputs of interest with their corresponding UTXOs. This structure can then be reused by both `createTx` and `RemoveTxOutputs`, avoiding multiple round-trips to the database for the same data and improving the performance of block processing.

Recommendations for optimising gas usage

ID	IF-FINDING-026
Severity	Informational
Impact	1 - Low
Exploitability	0 - None
Type	Implementation
Status	Patched Without Reaudit

Involved artifacts

For Reactor & Skyline:

- apex-bridge-smartcontracts/contracts

Description

The following finding describes a set of miscellaneous possible gas optimisations found during the audit.

Multiple accesses of a storage field should use a local variable cache

The `chainTokenQuantity` mapping in the `Claims` contract is accessed multiple times in functions `_submitClaimsBRC()`, `_submitClaimsRRC()`, and `_submitClaimsBEFC()` (see example here). To improve gas efficiency, the value should be cached in memory before operations and written back to storage only once. This caching pattern is already implemented in the `defund()` and `updateChainTokenQuantity()` functions.

One example is in function `_submitClaimsBEFC()`, where the `confirmedTransactions` mapping is accessed during each iteration of the signed batch's transaction nonces loop. The loop in `_submitClaimsBEFC()` could be implemented like this:

```
uint256 _currentAmount = chainTokenQuantity[chainId];
for (uint64 i = _firstTxNonce; i <= _lastTxNouce; i++) {
  ConfirmedTransaction storage _ctx = confirmedTransactions[chainId][i];
  uint8 _txType = _ctx.transactionType;
  if (_txType == 0) {
    _currentAmount += _ctx.totalAmount;
  } else if (_txType == 1) {
    if (_ctx.retryCounter < MAX_NUMBER_OF_DEFUND_RETRIES) {
      uint64 nextNonce = ++lastConfirmedTxNonce[chainId];
      confirmedTransactions[chainId][nextNonce] = _ctx;
      confirmedTransactions[chainId][nextNonce].nonce = nextNonce;
      confirmedTransactions[chainId][nextNonce].retryCounter++;
    } else {
      _currentAmount += _ctx.totalAmount;
      emit DefundFailedAfterMultipleRetries();
    }
  }
}

chainTokenQuantity[chainId] = _currentAmount;
```

Another example is in function `setChainAdditionalData()` where the loop over the `chains` in storage could be implemented as:

```
for (uint i = 0; i < chains.length; i++) {
  Chain memory chain = chains[i];
```

```
  if (chain.id == _chainId) {
    chain.addressMultisig = addressMultisig;
    chain.addressFeePayer = addressFeePayer;
    chains[i] = chain;
    break;
  }
}
```

Reorder contract variables to minimise number of storage slots used

In `Validators.sol` the fields could be re-arranged from:

```
address private upgradeAdmin;

// slither-disable too-many-digits
address constant PRECOMPILE = 0x0000000000000000000000000000000000002050;
uint32 constant PRECOMPILE_GAS = 50000;
address constant VALIDATOR_BLS_PRECOMPILE = 0x0000000000000000000000000000000000002060;
uint256 constant VALIDATOR_BLS_PRECOMPILE_GAS = 50000;

address private bridgeAddress;

// BlockchainId -> ValidatorChainData[]
mapping(uint8 => ValidatorChainData[]) private chainData;
// validator address index(+1) in chainData mapping
mapping(address => uint8) private addressValidatorIndex;

uint8 public validatorsCount;
```

to:

```
address private upgradeAdmin;

// slither-disable too-many-digits
address constant PRECOMPILE = 0x0000000000000000000000000000000000002050;
address constant VALIDATOR_BLS_PRECOMPILE = 0x0000000000000000000000000000000000002060;

address private bridgeAddress;

// BlockchainId -> ValidatorChainData[]
mapping(uint8 => ValidatorChainData[]) private chainData;
// validator address index(+1) in chainData mapping
mapping(address => uint8) private addressValidatorIndex;

uint32 constant PRECOMPILE_GAS = 50000;
uint32 constant VALIDATOR_BLS_PRECOMPILE_GAS = 50000;
uint8 public validatorsCount;
```

Please note also that the type for the field `VALIDATOR_BLS_PRECOMPILE_GAS` is changed from `uint256` to `uint32`, just as `PRECOMPILE_GAS`. These changes would save two storage slots, because the last three fields can be packed in a single slot.

Avoid unnecessary unsigned integer casting

In general it is more gas efficient to use `uint256` by default for most scenarios, especially for function parameters, return values, and local variables. This is because the EVM operates with 32-byte words and automatically converts smaller integers to `uint256` when they are used, which requires additional gas. As an exception, using smaller integer types is justified when the variables can be packed together in storage and will not require frequent conversions.

For example, the signatures of functions `updateTimeoutBlocksNumber()` and `updateMaxNumberOfTransactions()` could be updated to:

```
function updateTimeoutBlocksNumber(uint256 _timeoutBlocksNumber)
function updateMaxNumberOfTransactions(uint256 _maxNumberOfTransactions)
```

Cache array length in local variable and check it in loop iterations

When looping through an array, checking in every iteration of the loop the length of an array stored in storage is inefficient because it requires reading from storage repeatedly. For example, the loop in function `setChainAdditionalData()` could be implemented alternatively like this:

```
uint256 chainsLength = chains.length;
for (uint i = 0; i < chainsLength; i++) {
  if (chains[i].id == _chainId) {
    chains[i].addressMultisig = addressMultisig;
    chains[i].addressFeePayer = addressFeePayer;
    break;
  }
}
```

Similarly, in function `setValidatorsChainData()` the existing local cache for the number of validator should be used in the loop:

```
for (uint i; i < validatorsCnt; i++) {
  ...
}
```

Use pre-increment instead of post-increment

The reason behind this is in the way pre- and post-increment are evaluated by the compiler. Post-increment returns the value before incrementing it, which means that 2 values are stored on the stack for usage. However, pre-increment on the other hand, evaluates the increment operation on the value and then returns it, which means that only one item needs to be stored on the stack.

This recommendation is already followed in most of the codebase, and we could only found maybe one more place where it could be implemented (in function `getBatchingTxsCount()` where the variables `counterConfirmedTransactions` and `txIndx` are incremented in the `while` loop):

```
while (counterConfirmedTransactions < txsToProcess) {
  if (confirmedTransactions[_chainId][txIndx].blockHeight >= timeoutBlock) {
    break;
  }

  ++counterConfirmedTransactions;
  ++txIndx;
}
```

Use named results when possible

The solidity compiler outputs more efficient code when the variable is declared in the return statement. This is a practice that the codebase already braoadly adheres to, but there are still a few places where it could be implemented, like for example in function `_bytes32ToBytesAssembly()`:

```
function _bytes32ToBytesAssembly(
  bytes32 input
) internal pure returns (bytes memory output) {
```

```
  output = new bytes(32);

  assembly {
    mstore(add(output, 32), input)
  }

  return output;
}
```

Use unchecked math where appropriate

Solidity uses checked math by default to protect against overflow/underflow, but there are some situations where overflow/underflow might be infeasible to occur, and unchecked math could be used instead to reduce gas usage. Already done in a couple of places in the codebase, we think that it would be possible to be more aggressive and use unchecked math in a few more places. For example, in function `addValidatorChainData()` subtracting 1 to `addressValidatorIndex[_addr]` can be unchecked since this function is called through the bridge contract, which in function `registerChainGovernance()`, checks that the message sender is already a validator:

```
unchecked {
  uint8 indx = addressValidatorIndex[_addr] - 1;
  chainData[_chainId][indx] = _data;
}
```

Gas refund for deleting confirmed transactions that have completed lifecycle

When bridging request claims, refund request claims and defund requests are confirmed by a quorum of validators, a confirmed transactions is added to the `confirmedTransactions` mapping. The confirmed transactions must remain stored until the lifecycle of the batch that includes them is completed (i.e., either the batch successfully execute on the destination chain, or fails or times out). Once the lifecycle of the batch has completed by processing a batch executed claim, or a batch execution failed claim and the corresponding refund request claims, and no more retries need to be attempted, the confirmed transactions included in the batch might not be needed anymore and could be deleted from the mapping. Deleting the confirmed transactions would have the benefit of receiving gas refunds, because storage space is freed up.

Error handling function not fully tested

ID	IF-FINDING-027
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

For Reactor & Skyline:

- cardano-infrastructure/indexer/block_syncer.go
- cardano-infrastructure/indexer/block_syncer_test.go

Description

The Go test coverage command `go test -coverprofile=coverage.out ./...` revealed that the `errorHandler` function lacks complete unit test coverage, specifically for the logic that checks errors and restarts the synchronization process.

Recommendation

The error handling function plays a critical role in ensuring the indexer can automatically restart after non-fatal errors, such as connection drops to the Cardano node. Therefore it should be covered by the tests.

Exposing a testing function in production code

ID	IF-FINDING-028
Severity	Informational
Impact	1 - Low
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

For Reactor & Skyline:

- apex-bridge/batcher/evm_chain_operations.go

Description

The codebase contains explicitly coded failure mechanisms intended for testing. The `getTTLFormatter()` function returns different formatters based on a `testMode` configuration parameter that deliberately cause batches to fail by setting their `TTL` (Time-To-Live) to 0.

Problem Scenarios

While the configuration is controlled, this presents several risks:

- **Maintainability issues**: Future developers may not understand the purpose of these failure modes, leading to confusion or bugs.
- **Confusion in documentation**: The code comments are inconsistent with implementation. For example, the comment for case 3 states "First batch 5 batches fail" (here) but the code actually fails odd-numbered batches up to 10 (here).

Recommendation

Our recommendation is to address this issue by:

- **Separating test and production code**: Move test-specific functionality to a separate testing package or use build constraints (`// +build test`) to exclude this code from production builds.
- **Improving documentation**: If this functionality must remain, clearly document its purpose, potential risks, and that it should never be enabled in production environments.
- **Adding runtime safeguards**: Consider adding runtime checks that prevent test modes from being activated in production environments

Batcher miscellaneous code concerns

ID	IF-FINDING-029
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

For Reactor:

- batcher/evm_chain_operations.go
- batcher/cardano_chain_operations.go

Description

The following finding describes a set of miscellaneous code concerns found during the audit:

- Unnecessary casting and memory allocation here. `DfmToWei()` already accepts `*big.Int` type.
- Use `go-ethereum`'s `Cmp` instead of writing it's implementation by hand (here).
- Using `int` instead of `uint` as data type for configuration fields that only can have positive values (here).
- `getNeededUtxos()` function (here) takes explicit parameters for `MaxUtxoCount`, `TakeAtLeastUtxoCount` and `UtxoMinAmount`. All of these mentioned function parameters belong to `CardanoChainConfig` which means that it is would be better to pass only config struct instead of separate fields. This will bring clarity to `getNeededUtxos()` function.
- Function `getCardanoData()` (here) has a misleading name since it is returning `[]eth.ValidatorChainData`

Systematic lack of documentation

ID	IF-FINDING-030
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Documentation
Status	Acknowledged

Involved artifacts

For Reactor & Skyline:

- cardano-infrastructure/indexer
- apex-bridge/batcher
- apex-bridge-smartcontracts/contracts

Description

The codebase suffers from a widespread lack of documentation. Functions are missing explanations of their purpose, parameters, return values, and error conditions. Critical preconditions and assumptions that the code relies on are not documented, and complex logic lacks explanatory comments. This documentation gap exists across multiple components, making the code difficult to understand, maintain, and audit.

Problem Scenarios

Poor documentation creates several significant risks, as maintenance becomes error-prone as developers may unknowingly violate undocumented assumptions, security vulnerabilities are more likely to be introduced during code changes, incident response is slowed as responders must first understand undocumented code behavior and onboarding new developers takes longer, reducing team productivity.

Recommendation

Implement a documentation standard requiring function headers that describe purpose, parameters, returns, error conditions, and security assumptions for all code components. Integrate documentation quality checks into your code review process, with particular emphasis on security-critical sections and code that contains implicit preconditions.

Appendix: E2E Tests Review

Executive Summary

This review focuses on the end-to-end (E2E) tests implemented for the Apex Bridge system. The tests are located in the following files:

- blade-apex-bridge/e2e-polybft/e2e/apex_bridge_test.go
- blade-apex-bridge/e2e-polybft/e2e/nexus_bridge_test.go
- blade-apex-bridge/e2e-polybft/e2e/testnet_bridge_test.go

The review involved reading the test code, analyzing its coverage, and running the tests on a virtual machine. The machine used for testing was a Virtual Ubuntu 22.04 x64 instance with 32 GB of memory and 8 Intel vCPUs. The results of the test runs, along with observations and recommendations, are detailed below.

Test Coverage Analysis

Functional Coverage

The E2E tests cover a wide range of functionalities for the Apex Bridge system, including:

- **Infrastructure and endpoint verification**:
  - Ensures that the bridge components (Prime, Vector, and Nexus chains) are properly set up and operational.
- **Bridge operations**:
  - Submitting bridge requests.
  - Funding and defunding operations for hot wallets.
  - UTxO consolidation to handle fragmented outputs.
  - Validation of requests, including invalid scenarios.
- **Scenario Variations**:
  - Sequential and parallel execution of bridge requests.
  - Bidirectional operations between chains (e.g., Prime <-> Vector, Prime <-> Nexus).
  - Large-scale stress tests with thousands of transactions.
  - Node failure scenarios, including partial validator outages.
  - Randomized operations to simulate real-world conditions.
- **State Verification**:
  - Ensures that the bridge states and source/destination chain balances are consistent after operations.

Coverage Gaps

While the tests provide extensive coverage, the following scenarios are not addressed and are recommended for future inclusion:

1. **Validator sync recovery**:
   - A scenario where a validator is completely out of sync or heavily delayed compared to others, verifying its ability to catch up and participate in the quorum.
2. **Network disconnection recovery**:
   - Simulating bridge validators losing connection to Prime or Vector nodes and verifying their ability to recover.
3. **Chain rollback handling**:
   - Testing how the bridge handles chain rollbacks on Prime or Vector.
4. **TTL expiration**:
   - Testing batch execution failures due to TTL (time-to-live) expiration.
5. **Bridge request transaction outputs**:

- For Cardano chains, test the scenarios where a bridge request is submitted in a transaction with multiple outputs to the bridge address.

Test Execution Results

Apex Bridge Tests

The Apex bridge tests were executed, with the following results:

- **Passed tests**:
  - Most tests passed without issues, demonstrating the robustness of the bridge under various scenarios.
- **Failed tests**:
  - `TestE2E_ApexBridge_ValidScenarios_BigTests` / `From_prime_to_vector_1000x_20min_90%`:
    - Timed out without reaching the 90% success rate of transactions (30710000000/31074000000).
  - `TestE2E_ApexBridge_ValidScenarios_BigTests` / `From_prime_to_vector_1000x_60min_90%`:
    - Timed out without reaching the 90% success rate of transactions (31105000000/31597000000).

Nexus Bridge Tests

All Nexus bridge tests passed successfully, demonstrating the stability of the bridge between Nexus and Prime chains under the tested scenarios.

Testnet Bridge Tests

The testnet bridge tests were not executed because they are designed to run against a live testnet network, which requires an extensive setup that is not documented. The lack of clear instructions for replicating the required environment made it impractical to validate these tests effectively.

Observations and Recommendations

Observations

- **Skyline mode:** The Skyline mode introduces bridging operations between Cardano and Prime chains, enabling the Apex bridge to support both native and wrapped tokens. The E2E test suite has been refactored to support both Reactor and Skyline bridge modes. Additionally, the E2E tests have been extended to cover the Skyline mode operations to validate its added functionalities.

  The Skyline mode tests were reviewed and executed using the skyline_bridge_test.go file from `blade-apex-bridge/e2e/e2e-polybft` directory. These tests cover the following scenarios:

  - **Valid scenarios**: Bridging, funding, defunding, and UTxO consolidation operations using either native or wrapped tokens.
  - **Invalid scenarios**: All invalid scenarios covered in Reactor mode are also tested in Skyline mode. Additionally, Skyline mode introduces a new invalid scenario that tests bridging requests with invalid operation fees, specifically for wrapped token transfers. Currently, this scenario is only covered for bridging tokens from Prime to Cardano chains.

    All the Skyline mode tests were successfully run on the same virtual machine as the Reactor mode tests.

- **Documentation issues**:

  There is a significant lack of documentation for the E2E tests, which limited the depth of the code review and made the test setup more challenging. Some tests, such as `TestE2E_ApexBridge_Fund_Defund`, are very long (nearly 500 lines) and involve complex concurrent operations, making them difficult to understand and maintain.

- **Lack of automation for test setup**:

  Setting up the tests requires significant manual effort to compile and prepare the necessary binaries. Below is a list of all the binaries that need to be compiled or pre-compiled to run the tests:

  - Cardano Node/CLI: v8.7.3
  - Vector Node/CLI: Beta v8.9.3.0.0.0.0.5
  - Vector Ogmios: v6.2.0
  - Ogmios: v6.0.3
  - Polygon-Edge, Blade, Vector: Ethernal-Tech/blade-apex-bridge
    - Commit used for the Reactor review: 66247f3
    - Commit used for the Skyline review: a3a5962
  - Apex-Bridge: Ethernal-Tech/apex-bridge:
    - Commit used for the Reactor review: adf0d72
    - Commit used for the Skyline review: e527566

  The lack of automation for this setup process significantly increases the effort required to run the tests and could be a barrier for new contributors or reviewers.

- **Test execution time**:

  Certain tests, especially large-scale stress tests, take an excessive amount of time to complete, which could hinder iterative development and debugging.

Recommendations

To improve the E2E test suite, the following recommendations are proposed:

1. **Expand test coverage**:
   - Add more scenarios for validator sync recovery, network disconnection recovery, chain rollback handling, TTL expiration, and bridge requests with multiple outputs to the bridge address.
   - Add invalid scenarios to Skyline mode tests to cover the cases where bridging requests have invalid bridging and/or operations fees in both directions between the Cardano and Prime chains.
   - Add stress tests to Skyline mode E2E tests.
2. **Improve documentation**:
   - Provide detailed documentation for each test, including its the expected behaviour and results.
   - Include a comprehensive guide for setting up the E2E tests environment.
3. **Refactor long tests**:
   - Break down long tests into smaller, and try avoiding using to much concurrent processes to improve readability and maintainability.

This review highlights the impressive strengths of the E2E test suite, which is very well architected and demonstrates a high level of engineering to implement tests of such complexity. The tests extensively and successfully cover the main features and edge cases of the bridge system, showcasing its correctness and robustness. All tests ran successfully, except for two stress tests that came very close to passing; these failures are not considered significant.

While the test suite is already highly effective, there is always room for improvement. Within the time constraints of this audit, we identified potential coverage gaps and provided recommendations to further enhance the suite's documentation, maintainability, and overall coverage.

Appendix: Fuzz testing getNeededUtxos()

In order to gain more confidence that nothing will break with `getNeededUtxos()` function, we have implemented the following fuzz test.

```
package batcher

import (
 "errors"
 "testing"

 "github.com/Ethernal-Tech/cardano-infrastructure/indexer"
)

func FuzzGetNeededUtxos(f *testing.F) {
 f.Add(uint64(1000), uint64(1000000), uint64(2000000), 1, 5, 2) // Base case
 f.Add(uint64(5), uint64(10), uint64(10), 0, 1, 1  // Minimum values
 f.Add(uint64(1000000), uint64(10000000), uint64(1000000), 50, 100, 20) // Larger values

 f.Fuzz(func(t *testing.T, lengthInputUtxos uint64, desiredAmount uint64, minUtxoAmount uint64,
↪  utxoCount int, maxUtxoCount int, takeAtLeastUtxoCount int) {
  // Ensure sane input values
  if maxUtxoCount <= 0 || maxUtxoCount > 100 {
   return // Skip invalid max UTXO counts
  }
  if takeAtLeastUtxoCount > maxUtxoCount {
   return // Skip invalid minimum UTXO counts
  }
  if utxoCount < 0 || utxoCount > maxUtxoCount {
   return // Skip if initial count negative or  exceeds max
  }
  if desiredAmount == 0 {
   return // Skip zero desired amount
  }
  if minUtxoAmount == 0 {
   return // Skip zero minimum amount
  }

  // Create test UTXOs with varying amounts
  inputUTXOs := make([]*indexer.TxInputOutput, 0, lengthInputUtxos)
  totalAvailable := uint64(0)

  // Generate UTXOs with different amounts
  for i := range lengthInputUtxos {
   amount := uint64(i+1) * minUtxoAmount
   utxo := &indexer.TxInputOutput{
    Input: indexer.TxInput{
     Hash:  indexer.Hash{},
     Index: uint32(i),
    },
    Output: indexer.TxOutput{
     Amount: amount,
    },
   }
   inputUTXOs = append(inputUTXOs, utxo)
```

```
  totalAvailable += amount
 }

 chosenUTXOs, err := getNeededUtxos(
  inputUTXOs,
  desiredAmount,
  minUtxoAmount,
  utxoCount,
  maxUtxoCount,
  takeAtLeastUtxoCount,
 )

 if err != nil {
  // Verify error cases
  if totalAvailable < desiredAmount+minUtxoAmount {
   if !errors.Is(err, errUTXOsCouldNotSelect) {
    t.Errorf("Expected errUTXOsCouldNotSelect when total available < desired amount + min
↪  UTXO, got %v", err)
   }
  } else if len(inputUTXOs) > maxUtxoCount {
   if !errors.Is(err, errUTXOsLimitReached) {
    t.Errorf("Expected errUTXOsLimitReached when input UTXOs > max count, got %v", err)
   }
  }
  return
 }

 // Verify the results
 if len(chosenUTXOs) == 0 {
  t.Error("Expected non-empty chosen UTXOs")
  return
 }

 if len(chosenUTXOs) > maxUtxoCount {
  t.Errorf("Chosen UTXOs count %d exceeds maxUtxoCount %d", len(chosenUTXOs), maxUtxoCount)
 }

 // Calculate total amount in chosen UTXOs
 totalChosen := uint64(0)
 for _, utxo := range chosenUTXOs {
  totalChosen += utxo.Output.Amount
 }

 // Verify minimum amount requirement
 if totalChosen < desiredAmount+minUtxoAmount && totalChosen != desiredAmount {
  t.Errorf("Total chosen amount %d is less than required %d", totalChosen,
↪  desiredAmount+minUtxoAmount)
 }

 // Verify takeAtLeastUtxoCount requirement when possible
 remainingSlots := maxUtxoCount - (len(chosenUTXOs) + utxoCount)
 neededForMinimum := takeAtLeastUtxoCount - len(chosenUTXOs)
 inputsLeft := len(inputUTXOs) - len(chosenUTXOs)

 // Check we meet takeAtLeastUtxoCount when we have room
 if neededForMinimum < remainingSlots && len(chosenUTXOs) < takeAtLeastUtxoCount &&
 ↪  len(inputUTXOs) >= takeAtLeastUtxoCount {
```

```
   t.Errorf("Chosen UTXOs count %d is less than takeAtLeastUtxoCount %d when enough room (%d)
↪  and inputs (%d) available",
    len(chosenUTXOs), takeAtLeastUtxoCount, remainingSlots, lengthInputUtxos)
  }

  // Check we use all inputs when they would fit and are needed
  if inputsLeft < remainingSlots && inputsLeft <= neededForMinimum {
   expected := len(inputUTXOs)
   actual := len(chosenUTXOs)
   if actual < expected {
    t.Errorf("Expected all %d inputs to be used when they fit in remaining slots (%d) and are
↪  needed for minimum (%d), but got %d",
     expected, remainingSlots, neededForMinimum, actual)
   }
  }

  // Verify UTXOs are unique
  seen := make(map[uint32]bool)
  for _, utxo := range chosenUTXOs {
   if seen[utxo.Input.Index] {
    t.Error("Duplicate UTXO found in chosen UTXOs")
   }
   seen[utxo.Input.Index] = true
  }
 })
}
```

Instructions to run it can be found on the following code snippet:

```
go test . -run=^$ -fuzz=FuzzGetNeededUtxos -fuzztime=1m
```

Appendix: Tests Review

Indexer Tests

We have run the tests from the Indexer package and the results of the coverage can be found in the snippet bellow.

```
ok      github.com/Ethernal-Tech/cardano-infrastructure/indexer                 6.425s
↪  coverage: 64.3% of statements
        github.com/Ethernal-Tech/cardano-infrastructure/indexer/db
        ↪  coverage: 0.0% of statements
ok      github.com/Ethernal-Tech/cardano-infrastructure/indexer/db/bbolt        0.923s
↪  coverage: 81.1% of statements
        github.com/Ethernal-Tech/cardano-infrastructure/indexer/db/leveldb
        ↪  coverage: 0.0% of statements
```

Bridge Smart Contracts Tests

We have run the tests in Bridge Smart contracts and found that coverage was very satisfying.

```
----------------------|----------|----------|----------|----------|----------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------------|----------|----------|----------|----------|----------------|
 contracts/           |    95.56 |    78.79 |    83.33 |    94.21 |                |
  Admin.sol           |    86.67 |    72.73 |       75 |    85.71 |       40,70,71 |
  Bridge.sol          |     93.1 |    77.42 |    83.33 |    90.12 |... 263,267,268 |
  Claims.sol          |    98.56 |    90.82 |    89.66 |    98.51 |    373,499,500 |
  ClaimsHelper.sol    |    86.67 |    69.23 |    78.57 |    85.19 |  97,98,107,108 |
  SignedBatches.sol   |    95.45 |    66.67 |    81.82 |    94.29 |        133,134 |
  Slots.sol           |    92.86 |    66.67 |       75 |       92 |          78,79 |
  Validators.sol      |    96.67 |       65 |     87.5 |    92.86 |    126,167,168 |
 contracts/interfaces/ |      100 |      100 |      100 |      100 |                |
  IBridge.sol         |      100 |      100 |      100 |      100 |                |
  IBridgeStructs.sol  |      100 |      100 |      100 |      100 |                |
----------------------|----------|----------|----------|----------|----------------|
All files             |    95.56 |    78.79 |    83.33 |    94.21 |                |
----------------------|----------|----------|----------|----------|----------------|
```

Batcher tests

Due to certain platform constraints, we were not able to run batcher tests.

Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for $< 3\%$; *medium* for 3-10%; *large* for 10-33%, *all* for $>33\%$. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	**Examples**
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

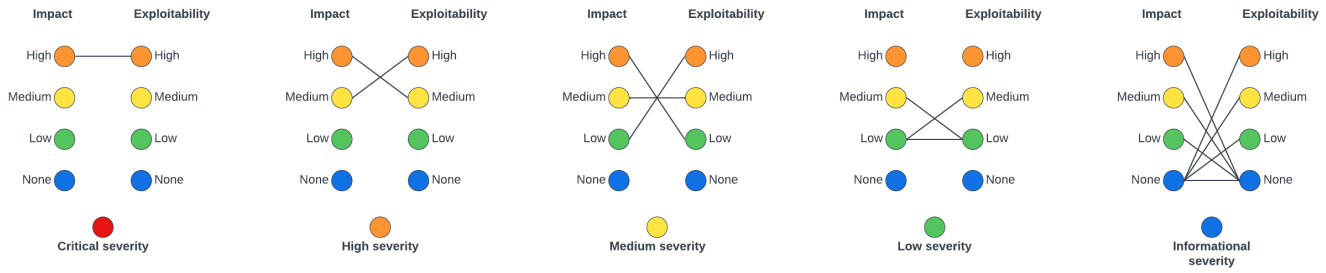

Figure 3: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

SeverityScore	Examples
Critical	Halting of chain via a submission of a specially crafted transaction
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an "as is" basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an "endorsement", "approval" or "disapproval" of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client's business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.