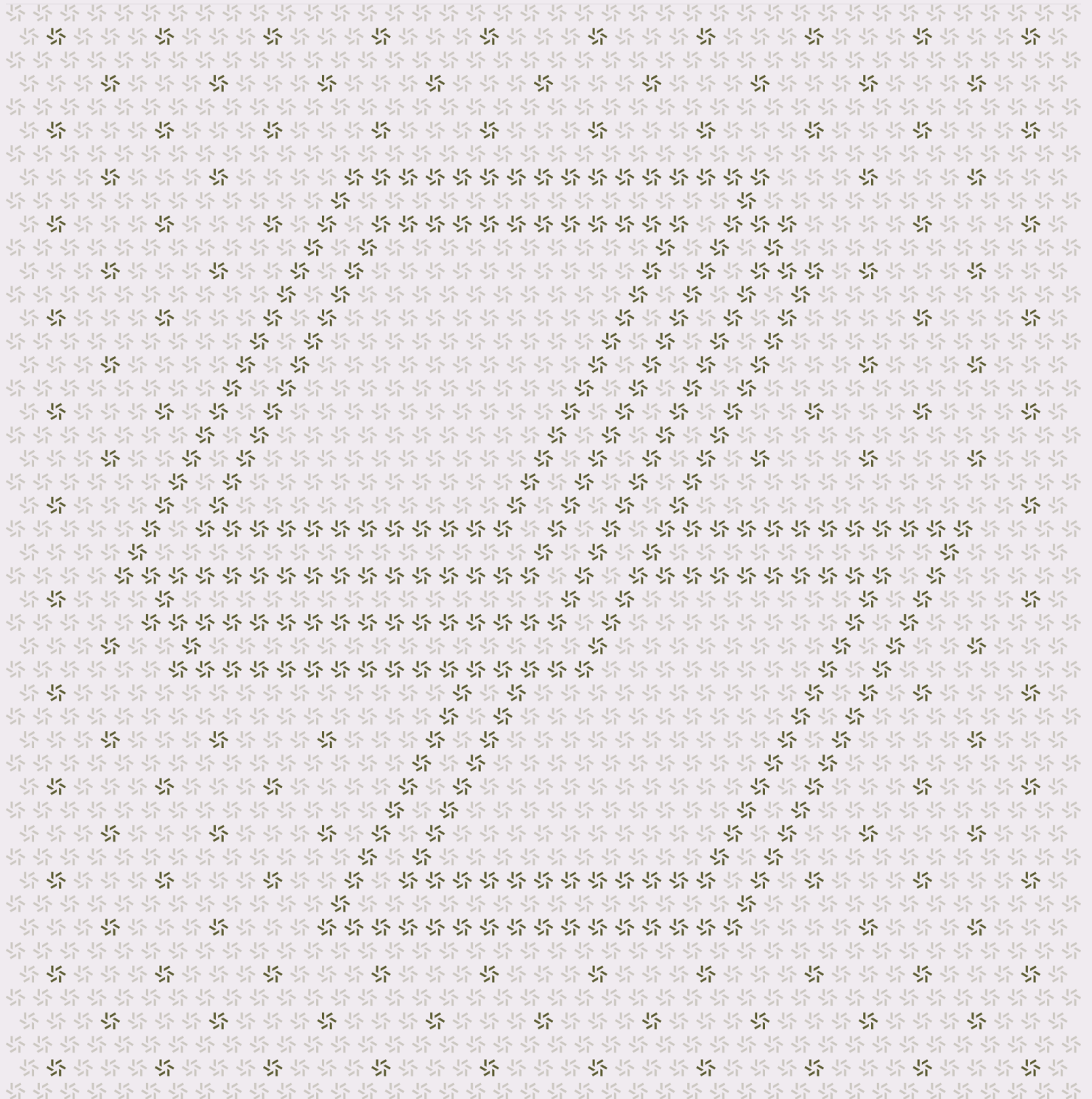


December 22, 2025

Jovay Relayer

Application Security Assessment



Contents

About Zellic	4
<hr/>	
1. Overview	4
1.1. Executive Summary	5
1.2. Goals of the Assessment	5
1.3. Non-goals and Limitations	5
1.4. Results	5
<hr/>	
2. Introduction	6
2.1. About Jovay Relayer	7
2.2. Methodology	7
2.3. Scope	9
2.4. Project Overview	9
2.5. Project Timeline	10
<hr/>	
3. Detailed Findings	10
3.1. Heap out-of-memory issue via highly compressible data	11
3.2. Missing access control in AdminGrpc API	14
3.3. Quadratic complexity in batch processing	15
3.4. Insufficient test coverage	18
3.5. Secrets exposed in logs	20
3.6. Incorrect address type in withdrawVault	22
3.7. Network I/O within database transaction	24
3.8. Missing bounds check in chunk deserialization	26

3.9.	Relayer enters infinite loop when block exceeds blob size limit	28
3.10.	Relayer centralization risks	30
3.11.	Missing size check in Zstd decompression	32
<hr data-bbox="488 525 1563 529"/>		
4.	Discussion	32
4.1.	Considerations on zstd compression	33
4.2.	Considerations on oracle transaction failure handling	33
<hr data-bbox="488 787 1563 791"/>		
5.	System Design	33
5.1.	Component: Rollup Batch Management Service (IRollupService)	34
5.2.	Component: L1 Listen Service (IL1ListenService)	35
5.3.	Component: Mailbox Service (IMailboxService)	37
5.4.	Component: Reliable Transaction Service (IReliableTxService)	38
5.5.	Component: Oracle Service (IOracleService)	40
<hr data-bbox="488 1228 1563 1232"/>		
6.	Assessment Results	41
6.1.	Disclaimer	42

About Zellic

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

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

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



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Ant Digital Technologies from November 28th to December 18th, 2025. During this engagement, Zellic reviewed Jovay Relayer's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

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

- Are there bugs that can cause financial losses, chain-data corruption, or service interruptions?
 - Does the IRollupService batch blob-limit validation correctly handle the EIP-4844 limit after serialization?
 - Is the usage of the zstd-compression algorithm appropriate and deterministic?
 - Are there security vulnerabilities in the IRollupAggregator.process method?
 - Are all transaction-state transitions in IReliableTxService handled correctly, ensuring transactions reach their final state?
-

1.3. Non-goals and Limitations

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

- EthBlobForkConfig, RollupEconomicStrategy, and GrowingBatchChunksMemCache, as they were excluded from the audit scope
- Front-end components
- Infrastructure relating to the project (Prover, Tracer, Verifier, and so on)
- Key custody

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

1.4. Results

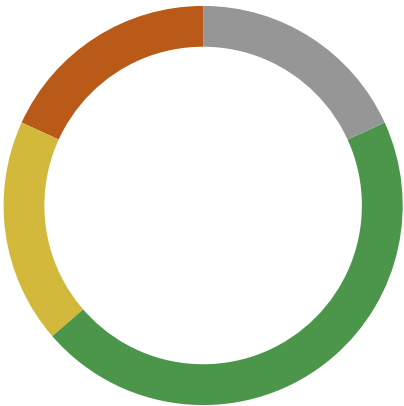
During our assessment on the scoped Jovay Relayer targets, we discovered 11 findings. No critical issues were found. Two findings were of high impact, two were of medium impact, five were of low

impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Ant Digital Technologies in the Discussion section (4.7).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	2
Low	5
Informational	2



2. Introduction

2.1. About Jovay Relayer

Ant Digital Technologies contributed the following description of Jovay Relayer:

Jovay is a high-performance, user-friendly Layer 2 scaling solution that aims to break through the scalability bottleneck of blockchain through innovative technologies while maintaining compatibility with the Ethereum ecosystem.

Jovay is tailored for real-world assets, applications, and users. Our commitment lies in establishing trust through data and AI, prioritizing the developer experience, facilitating asset bridging for interchain liquidity, conducting asset tokenization, and integrating value.

2.2. Methodology

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

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

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

Architecture risks. This encompasses potential hazards originating from the blueprint of a system, which involves its core validation mechanism and other architecturally significant constituents influencing the system's fundamental security attributes, presumptions, trust mode, and design.

Arithmetic issues. This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

Implementation risks. This encompasses risks linked to translating a system's specification into practical code. Constructing a custom system involves developing intricate on-chain and off-chain elements while accommodating the idiosyncrasies and challenges presented by distinct programming languages, frameworks, and execution environments.

Availability. Denial-of-service attacks are another leading issue in custom systems. Issues including but not limited to unhandled panics, unbounded computations, and incorrect error handling can potentially lead to consensus failures.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no

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

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

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped targets itself. These observations — found in the Discussion (4. 7) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Jovay Relay Targets

Type	Java
Platform	EVM-compatible
Target	jovay-relayer
Repository	https://github.com/jovaynetwork/jovay-relayer
Version	98bbdb8392072866e0d6a7c0a4cc6425fb1cdbc8
Programs	relayer-app/* relayer-commons/*

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
✈ Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Michael Guarino
✈ Engineer
michael@zellic.io ↗

Jisub Kim
✈ Engineer
jisub@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 28, 2025 Kick-off call

November 28, 2025 Start of primary review period

December 18, 2025 End of primary review period

3. Detailed Findings

3.1. Heap out-of-memory issue via highly compressible data

Target	RollupAggregator, RollupUtils		
Category	Business Logic	Severity	High
Likelihood	High	Impact	High

Description

The RollupAggregator accumulates chunks in memory until a batch is committed. The batch-commit condition only checks the *compressed* data size against the blob limit, with no restriction on the *raw (uncompressed)* data size. This creates a dangerous gap: if data compresses extremely well, the raw data can grow unbounded while the compressed size stays under the limit.

An attacker can exploit this by sending highly compressible data on L2 (e.g., repeated null bytes \x00), which compresses at ratios of 1000:1 or higher with zstd. Since the batch commit is only triggered when the compressed size exceeds the limit, raw data continues to accumulate in memory indefinitely.

The problematic flow is as follows:

1. RollupAggregator.process()
2. buildL2NextChunk()
3. getGrowingBatchInfo()
4. new ChunksPayload(chunkList).serialize()
5. RollupUtils.serializeChunks(chunks)

The serializeChunks method creates a new ByteArrayOutputStream and copies all chunk data:

```
// RollupUtils.java
public static byte[] serializeChunks(List<Chunk> chunks) {
    var rawChunksStream = new ByteArrayOutputStream();
    var streamToWrite = new DataOutputStream(rawChunksStream);
    chunks.forEach(
        chunk -> {
            var raw = chunk.serialize();
            try {
                streamToWrite.writeInt(raw.length);
                streamToWrite.write(raw);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    );
}
```

```

    }
}

);
var rawChunks = rawChunksStream.toByteArray(); // Creates a copy of the
buffer
streamToWrite.close();
return rawChunks;
}

```

The batch-commit check only triggers when compressed size exceeds the limit:

```

// RollupAggregator.GrowingBatchInfo
public int getOverBatchBlobLimitFlag(int batchCommitBlobSizeLimit) {
    if (batchVersion == BatchVersionEnum.BATCH_V1) {
        var res = getPayloadSize() + BlobsDaData.DA_DATA_META_LEN_SIZE
        - batchCommitBlobSizeLimit * BlobsDaData.CAPACITY_BYTE_PER_BLOB;
        if (res >= 0) {
            // Only compresses when raw size exceeds limit
            res = getCompressedDataSize() - batchCommitBlobSizeLimit
            * BlobsDaData.CAPACITY_BYTE_PER_BLOB;
        }
        return res;
    }
    // [...]
}

```

If raw data is 2 GB but compresses to 100 KB, res becomes negative, and chunks continue to accumulate.

Furthermore, memory issues compound because of the following:

1. The `ByteArrayOutputStream.toByteArray()` function copies the entire buffer, so heap memory usage could be double.

```

/**
 * Creates a newly allocated byte array. Its size is the current
 * size of this output stream and the valid contents of the buffer
 * have been copied into it.
 *
 * @return the current contents of this output stream, as a byte
 *         array.
 * @see    java.io.ByteArrayOutputStream#size()
 */
public synchronized byte[] toByteArray() {
    return Arrays.copyOf(buf, count);
}

```

2. Each `Chunk.serialize()` also calls `toByteArray()`.
3. And `zstd` compression requires additional memory allocation.

For N bytes of raw data, peak memory usage could theoretically be **$2.5 N$ or more**.

Impact

The relayer crashes with an out-of-memory (OOM) error, halting all rollup operations. The attack is inexpensive to execute due to low L2 gas costs and can be repeated persistently.

Recommendations

Consider adding a hard limit on raw (uncompressed) data size before checking compressed size to force batch commits when memory usage becomes excessive.

Remediation

This issue has been acknowledged by Ant Digital Technologies, and a fix was implemented in commit [effc4d57](#).

3.2. Missing access control in AdminGrpc API

Target	AdminGrpcService		
Category	Protocol Risks	Severity	High
Likelihood	Medium	Impact	High

Description

The AdminGrpcService exposes multiple sensitive admin APIs via gRPC without any authentication or authorization mechanisms. The service relies solely on network-level security (i.e., the machine running the relayer is on an internal network).

If an attacker gains network access to the machine running the relayer (e.g., through a compromised internal network or misconfigured firewall), they could

- withdraw funds from the vault to arbitrary addresses via `withdrawFromVault()`
- manipulate gas-price configurations via `updateGasPriceConfig()`
- modify rollup economic parameters via `updateFixedProfit()` or `updateTotalScala()`

Impact

An attacker with network access could withdraw vault funds or disrupt relayer operations.

Recommendations

Consider implementing authentication and authorization for the gRPC service.

Remediation

This issue has been acknowledged by Ant Digital Technologies, and a fix was implemented in commit [42f9704c](#).

3.3. Quadratic complexity in batch processing

Target	RollupAggregator		
Category	Optimization	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

When `RollupAggregator.process()` is called during L2 \rightarrow L1 batch creation, the entire accumulated batch data is reserialized and compressed for every block processed. This results in $O(N^2)$ computational complexity, where N is the number of blocks in the batch.

The block processing pattern is as follows:

- Block 1: Serialize one chunk.
- Block 2: Serialize two chunks (including block 1).
- Block 3: Serialize three chunks.
- Block N : Serialize N chunks.

This pattern continues for each subsequent block, and the total operations are $1 + 2 + \dots + N = N(N+1)/2$, which means $O(N^2)$.

The issue occurs in the following call chain:

```
// RollupAggregator.java
public BigInteger process(BasicBlockTrace blockTrace) {
    // [...]
    var growingBatchInfo = getGrowingBatchInfo(buildingChunk);
    var overBatchBlobLimitFlag =
        growingBatchInfo.getOverBatchBlobLimitFlag(rollupConfig.getBatchCommitBlobSizeLimit());
    // [...]
}

private GrowingBatchInfo getGrowingBatchInfo(ChunkWrapper currChunk) {
    growingBatchChunks.checkAndFill(currChunk, rollupRepository);
    var chunkList = growingBatchChunks.copy();
    chunkList.add(currChunk.getChunk());
    return new GrowingBatchInfo(
        getCurrBatchVersion(getStartBlockTimestampForBatch(currChunk.getBatchIndex())),
        new ChunksPayload(chunkList).serialize() // Re-serializes ALL
        chunks
    );
}
```

```
);  
}
```

The compression check also triggers full compression on every call when the raw size exceeds the limit:

```
// RollupAggregator.GrowingBatchInfo  
public int getOverBatchBlobLimitFlag(int batchCommitBlobSizeLimit) {  
    if (batchVersion == BatchVersionEnum.BATCH_V1) {  
        var res = getPayloadSize() + BlobsDaData.DA_DATA_META_LEN_SIZE  
        - batchCommitBlobSizeLimit * BlobsDaData.CAPACITY_BYTE_PER_BLOB;  
        if (res >= 0) {  
            res = getCompressedDataSize() - batchCommitBlobSizeLimit  
            * BlobsDaData.CAPACITY_BYTE_PER_BLOB;  
        }  
        return res;  
    }  
    // [...]  
}  
  
private int getCompressedDataSize() {  
    if (compressedDataSize == -1) {  
        compressedDataSize  
        = this.batchVersion.getDaCompressor().compress(rawPayload).length  
        + BlobsDaData.DA_DATA_META_LEN_SIZE;  
    }  
    return compressedDataSize;  
}
```

While `compressedDataSize` is cached within a single `GrowingBatchInfo` instance, a new instance is created for each block processed, negating the caching benefit.

This quadratic behavior is exacerbated by the highly compressible data, where highly compressible data can accumulate many blocks before triggering a batch commit. (Please see [Finding 3.1](#) for more detail.)

Impact

As the number of blocks per batch increases, relayer performance degrades rapidly.

Recommendations

Consider implementing incremental size tracking to avoid reserialization on every block.

Remediation

This issue has been acknowledged by Ant Digital Technologies, and a fix was implemented in commit [eb2a7465](#).

3.4. Insufficient test coverage

Target	Project-wide		
Category	Protocol Risks	Severity	Medium
Likelihood	N/A	Impact	Medium

Description

In our opinion, many of the findings in this report could have been caught before the audit stage with a more complete test suite, featuring unit tests of both positive and negative scenarios for every function, integration tests to help with identifying bugs associated with more complex state changes, and tests for adversarial account scenarios.

When building a complex contract with multiple moving parts (state changes) and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios on every function that touches the contract's state.

Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts and all functions — not just surface-level functions. It is important to test the invariants required for ensuring security and also verify any mathematical properties from the project's specification. Additionally, testing cross-chain function calls and transfers is recommended to ensure the desired functionality.

Impact

Good test coverage has multiple effects:

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in the product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point may seem contradictory given the time investment to create and maintain tests. However, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks other code if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence.

Tests have your back here. They are an excellent indicator that the existing functionality was most likely not broken by a change to the code. Without a comprehensive test suite, the above benefits are lost. This increases the likelihood of bugs and vulnerabilities going unnoticed until after deployment, which can be costly and damaging to the project's reputation.

Recommendations

We recommend building a rigorous test suite that includes all components, including relayer end-to-end tests to ensure that the system operates securely and as intended.

Remediation

This issue has been acknowledged by Ant Digital Technologies, and a fix was implemented in commit [35e75aae](#).

3.5. Secrets exposed in logs

Target	PrefixedDynamicConfig		
Category	Code Maturity	Severity	Medium
Likelihood	Low	Impact	Low

Description

The PrefixedDynamicConfig class defines initCacheFromStorage, which logs config values upon initialization. Configs contain sensitive values including API keys, which will then be exposed in logs.

```
private void initCacheFromStorage() {
    var configs = systemConfigRepository.getPrefixedSystemConfig(prefix
        + "@");
    for (var entry : configs.entrySet()) {
        configCacheMap.putIfAbsent(entry.getKey(), entry.getValue());
    }
    log.info("Init dynamic config cache for {} from storage with: {}", prefix,
        JSON.toJSONString(configs)); // here
}
```

With the debug flag enabled, sensitive values are also logged by persistCurrConfig:

```
private void persistCurrConfig() {
    try {
        if (!persistLock.tryLock(0, persistInterval, TimeUnit.SECONDS)) {
            return;
        }
        for (var entry : configCacheMap.entrySet()) {
            if (ObjectUtil.isNull(entry.getValue())) {
                configCacheMap.remove(entry.getKey());
                log.error("Remove null value from config cache map: {}",
                    entry.getKey());
                continue;
            }
            log.debug("Persist current config: {} - {}", entry.getKey(),
                entry.getValue()); // here
            systemConfigRepository.setSystemConfig(entry.getKey(),
                entry.getValue());
        }
    }
}
```

```
log.info("Persist current config for {} with {} entries successfully",
prefix, configCacheMap.size());
} catch (Throwable t) {
    log.error("Persist current config failed: ", t);
}
```

Impact

Exposure of Etherscan or Owlracle API keys to an attacker can lead to denial of service by exhausting rate limits.

Recommendations

Avoid logging any sensitive values. Consider modifying affected areas of code to instead mask values of sensitive fields or refrain from logging entirely.

Remediation

This issue has been acknowledged by Ant Digital Technologies, and a fix was implemented in commit [89adfc9f](#).

3.6. Incorrect address type in withdrawVault

Target	L2Client		
Category	Coding Mistakes	Severity	Low
Likelihood	Low	Impact	Low

Description

In the `L2Client.withdrawVault()` function, the `account` parameter is passed as `Utf8String` instead of the proper `Address` type. This means no Ethereum address-format validation is performed, potentially allowing funds to be sent to invalid addresses.

```
// L2Client.java
@Retryable(retryFor = {TxSimulateException.class,
    ClientConnectionException.class, SocketException.class,
    SocketTimeoutException.class}, backoff = @Backoff(delay = 300),
    notRecoverable = Exception.class)
public TransactionInfo withdrawVault(String account, BigInteger amount) {
    log.info("start sending tx to withdraw specific ETH: {} from vault contract
to account: {} with retry {}", amount, account,
        ObjectUtil.isNull(RetrySynchronizationManager.getContext())
        ? 0 : RetrySynchronizationManager.getContext().getRetryCount());
    var function = new Function(
        L2CoinBase.FUNC_WITHDRAW,
        ListUtil.toList(
            new Utf8String(account), // <--- Should be Address type
            new Uint256(amount)
        ),
        Collections.emptyList()
    );
    // [...]
}
```

Impact

If an invalid address string is passed (e.g., malformed hex, incorrect length, or non-checksummed address when required), the transaction may fail at the contract level or send funds to an unrecoverable address.

Recommendations

Consider using the proper `Address` type instead of `Utf8String` to ensure address format validation before transaction submission.

Remediation

This issue has been acknowledged by Ant Digital Technologies, and a fix was implemented in commit [48e3cb70](#).

3.7. Network I/O within database transaction

Target	ReliableTxServiceImpl		
Category	Code Maturity	Severity	Low
Likelihood	Low	Impact	Low

Description

In `ReliableTxServiceImpl`, the `processL1NotFinalizedTx()` and `processL2NotFinalizedTx()` methods wrap their processing logic inside `transactionTemplate.execute()`. Within this transaction scope, network RPC calls to Ethereum nodes are made (e.g., `l1Client.queryTx()`, `queryTxReceipt()`, `sendRawTx()`).

```
// ReliableTxServiceImpl.java
@Override
public void processL1NotFinalizedTx() {
    var reliableTransactions =
        rollupRepository.getNotFinalizedReliableTransactions(ChainTypeEnum.LAYER_ONE,
            processBatchSize);
    // [...]
    for (ReliableTransactionDO tx : reliableTransactions) {
        try {
            transactionTemplate.execute(
                new TransactionCallbackWithoutResult() {
                    @Override
                    protected void
                    doInTransactionWithoutResult(TransactionStatus status) {
                        processL1PendingTx(tx); // Contains network I/O
                    }
                }
            );
        } catch (Exception e) {
            // [...]
        }
    }
}
```

See inside `processL1PendingTx()`:

```
// ReliableTxServiceImpl.java
private void processL1PendingTx(ReliableTransactionDO tx) {
```

```
var transaction = l1Client.queryTx(tx.getLatestTxHash()); // Network I/O
// [...]
TransactionReceipt receipt
= l1Client.queryTxReceipt(tx.getLatestTxHash()); // Network I/O
// [...]
EthSendTransaction sendResult = l1Client.sendRawTx(tx.getRawTx()); //
Network I/O
// [...]
}
```

If the Ethereum node response is delayed or the network is slow, the database connection remains held for the duration of the network call. This is an antipattern because database connections are a limited resource.

Impact

Network delays could lead to database connection time-outs or, in extreme cases, connection-pool exhaustion.

Recommendations

Consider separating network calls from database transactions by performing network I/O before or after the transaction scope and only including database updates within the transaction.

Remediation

This issue has been acknowledged by Ant Digital Technologies, and a fix was implemented in commit [4aad872d](#).

3.8. Missing bounds check in chunk deserialization

Target	Chunk, BlockContext		
Category	Code Maturity	Severity	Low
Likelihood	Low	Impact	Low

Description

The `Chunk.deserializeFrom()` and `BlockContext.deserializeFrom()` methods read from input byte arrays without verifying the array length. These methods increment an offset while reading, and if the input data is malformed or truncated, the underlying `System.arraycopy` in `BytesUtils` will throw an `IndexOutOfBoundsException`.

```
// Chunk.java
public static Chunk deserializeFrom(byte[] raw) {
    Chunk chunk = new Chunk();

    int offset = 0;
    chunk.setNumBlocks(BytesUtils.getUInt8(raw, offset++));

    List<BlockContext> blockContexts = new ArrayList<>();
    for (int i = 0; i < chunk.getNumBlocks(); i++) {
        // No check if offset exceeds raw.length
        blockContexts.add(BlockContext.deserializeFrom(
            ArrayUtil.sub(raw, offset, offset
                += BlockContext.BLOCK_CONTEXT_SIZE)
        ));
    }
    chunk.setBlocks(blockContexts);

    if (offset < raw.length - 1) {
        chunk.setL2Transactions(ArrayUtil.sub(raw, offset, raw.length));
    }

    return chunk;
}
```

```
// BlockContext.java
public static BlockContext deserializeFrom(byte[] raw) {
    BlockContext context = new BlockContext();
```

```
int offset = 0;
context.setSpecVersion(BytesUtils.getUInt32(raw, offset));
offset += 4;

context.setBlockNumber(BytesUtils.getUInt64(raw, offset));
offset += 8;
// ... continues reading without bounds checking

return context;
}
```

If malformed chunk data from an untrusted source is processed, the relayer thread may crash with an unexpected exception. While the current architecture ensures chunk data originates from trusted L2 block traces, this could become a vulnerability if the data source is compromised or the deserialization is used in a different context.

Impact

Processing malformed chunk data causes an uncaught exception, which could crash the processing thread.

Recommendations

Consider adding explicit bounds checking before reading from byte arrays to provide clearer error messages and prevent unexpected exceptions.

Remediation

This issue has been acknowledged by Ant Digital Technologies, and a fix was implemented in commit [175c7091](#).

3.9. Relayer enters infinite loop when block exceeds blob size limit

Target	RollupAggregator		
Category	Business Logic	Severity	Low
Likelihood	Low	Impact	Low

Description

In the `RollupAggregator.process()` function, when a single block at chunk index 0 exceeds the blob size limit (`batchCommitBlobSizeLimit`), the relayer throws an `InvalidBatchException` and cannot recover.

```
// RollupAggregator.java
} else if (ObjectUtil.equal(currChunkStartBlockNumber,
    currHeight) && overBatchBlobLimitFlag > 0) {
    if (nextChunkIndex == 0) {
        throw new InvalidBatchException("The first chunk of batch#{ } only
            contains one block#{ }, and already over the blobs capacity",
            nextBatchIndex, currChunkStartBlockNumber);
    }
    // [...]
}
```

This exception propagates up to `RollupServiceImpl.pollL2Blocks()`, which is wrapped inside `transactionTemplate.execute()`:

```
// RollupServiceImpl.java
transactionTemplate.execute(
    new TransactionCallbackWithoutResult() {
        @Override
        protected void doInTransactionWithoutResult(TransactionStatus status)
        {
            // [...]
            var batchThisBlockBelongsTo =
                rollupAggregator.process(blockTrace);
            // [...]
            rollupRepository.updateRollupNumberRecord(
                ChainTypeEnum.LAYER_TWO,
                RollupNumberRecordTypeEnum.BLOCK_PROCESSED,
                currHeight
            );
        }
    });
```

```
    }  
  }  
);
```

When the exception occurs, the transaction rolls back, and the `BLOCK_PROCESSED` height is never updated. In the next execution cycle, `pollL2Blocks()` fetches the same problematic block height from the database, causing the relayer to enter an infinite loop where it repeatedly attempts to process the same oversized block.

The blob size limit is fetched from the L1 contract configuration, while L2 block size limits are controlled by the L2 Sequencer. Since these configurations are not automatically synchronized, a simple parameter change on either side can trigger this scenario.

Impact

The relayer becomes permanently stuck, halting batch commits and proof submissions to L1. L2 state transitions cannot be finalized on L1 until manual intervention.

Recommendations

Consider handling oversized blocks gracefully instead of throwing an exception, ensuring the processed height is always updated to prevent infinite loops. Additionally, ensure that L1 blob size limits are synchronized with L2 block size configurations.

Remediation

This issue has been acknowledged by Ant Digital Technologies.

The client responded that this scenario is outside the intended design scope, noting that the Sequencer is designed not to produce blocks exceeding the blob size limit. Additionally, they clarified that an internal alerting system is in place so that, if such an incident were to occur, the technical team can intervene manually as part of established operational procedures.

3.10. Relayer centralization risks

Target	System architecture		
Category	Protocol Risks	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The current deployment architecture has centralization characteristics related to operator diversity and trust concentration.

The first characteristic is **relayer operator centralization**.

The relayer supports active-active (multiactive) deployment through the DistributedTaskEngine, which is responsible for distributing and executing tasks across different relayer nodes. Multiple relayer instances are already running in an active-active configuration in the production environment, which provides operational redundancy and fault tolerance.

However, the relayer infrastructure is operated solely by the Ant Digital Technologies, which creates organizational centralization.

The second observation is **limited MsgOracle operator diversity**.

As of the time of writing, there are five MsgOracle nodes operated by two organizations: Jovay (3 MsgOracle nodes) and ZAN (2 nodes). According to the design documentation, MsgOracle will introduce additional external trusted institutions in future phases.

The MsgOracle node count is managed by a multi-signature-like MsgOracle contract, allowing for future expansion without code changes.

In the current state, as of the time of writing,

- five nodes are operated by two organizations (Jovay and ZAN),
- the multi-sig contract allows for adding external institutions, and
- more operators are planned to be onboarded in phases.

Impact

While the active-active deployment mitigates availability risks, the remaining centralization creates organizational trust concentration: relayer infrastructure is operated by a single organization, and MsgOracle is currently operated by only two organizations.

Recommendations

Consider onboarding external trusted institutions to operate MsgOracle nodes and publishing a decentralization roadmap. Additionally, consider expanding the relayer operator set beyond the Ant Digital Technologies to further reduce organizational centralization.

Remediation

This issue has been acknowledged by Ant Digital Technologies.

The Ant Digital Technologies will expand the MsgOracle operator set in phases by onboarding independent third-party operators (institutions), with the goal of reducing centralization risk and improving fault tolerance.

3.11. Missing size check in Zstd decompression

Target	BlobsDaData		
Category	Code Maturity	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The Zstd decompression logic in the batch payload deserialization does not validate the dataLen field before decompression.

Impact

The practical impact is minimal due to physical blob size limitations. Without explicit validation, a malformed dataLen field could cause out-of-bound read or specially crafted compressed data could cause excessive memory allocation during decompression in edge cases or future implementations.

Recommendations

Consider adding explicit dataLen bounds checking before decompression and setting a hard upper limit on decompressed data size as a defensive measure.

Remediation

This issue has been acknowledged by Ant Digital Technologies, and a fix was implemented in commit [3a62a67b](#).

4. Discussion

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

4.1. Considerations on zstd compression

Generally, zstd compression produces deterministic output across x86, x64, and ARM architectures. A known nondeterminism issue existed when using `ZSTD_compressCCtx` (see issue [#1241 ↗](#)), but this option is not used by default and the issue has since been fixed.

A potential decompression bomb attack (similar to Netty's issue [#14004 ↗](#)) could theoretically cause a JVM OOM issue during decompression. However, we consider this a low practical risk in this system. While the `dataLen` field in `toBatchPayload` is a `uint24` (max ~16 MB), the actual blob capacity is only $31 * 4096 \approx 124$ KB per blob. Since compressed data originates from EIP-4844 blobs, the maximum decompressed size is inherently bounded.

4.2. Considerations on oracle transaction failure handling

We noted a design consideration in the oracle feeding transaction handling logic. When a transaction is lost or not found on-chain, the relayer currently marks it as `TX_SUCCESS`, which could cause a state inconsistency between the on-chain oracle data and the relayer's internal state. This may complicate debugging and monitoring.

This is not a security vulnerability, as the subsequent oracle update cycle will submit a new transaction regardless, ensuring that the oracle data does not remain stale. However, improving the transaction status handling would enhance system observability and reliability.

The team has acknowledged this design point and plans to address it in the next version.

5. System Design

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

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

5.1. Component: Rollup Batch Management Service (IRollupService)

Description

The Rollup Service is the core component orchestrating the complete rollup life cycle (L2 block polling, batch aggregation, L1 commitment, proof generation, L1 verification). It polls L2 blocks and retrieves block traces, aggregates blocks into chunks and chunks into batches, commits batches to the L1 rollup contract using EIP-4844 blob transactions, requests proofs from prover services (TEE/ZK), and submits proof-verification transactions to L1. The service manages commitment windows to control in-flight batches/proofs, enforces economic strategies to avoid committing during high gas prices, and handles resubmissions when necessary.

The implementation uses `CompletableFuture` for parallel block-trace fetching (default 10s time-out per block) and implements windowing (default 12 batches) to process multiple batches concurrently while maintaining ordering. It queries the L1 rollup contract for the last committed/verified indexes, applies `RollupAggregator` to process block traces, coordinates with `L2MsgFetcher` for cross-layer messages, and uses `ProverControllerClient` for async proof requests. `RollupThrottle` checks L1 transaction pool congestion before submitting. All state is maintained in database with Spring transactions ensuring atomic updates.

Invariants

- **Batch ordering.** Batch N+1 can only be committed after batch N is committed.
- **Proof ordering.** TEE/ZK proof for batch N can only be submitted after batch N is committed.
- **Sequential block processing.** L2 blocks must be processed in order without gaps.
- **Batch finality.** Once a batch is committed to L1, its contents are immutable.
- **Parent-child relationship.** Each batch must reference the correct parent batch hash.
- **Anchor batch genesis.** Batch 0 must be set as an anchor before processing begins.
- **Proof availability.** A proof can only be committed after the prover confirms it is ready.
- **Window constraints.** At most `batchCommitWindowsLength` batches can be in-flight.
- **Economic strategy compliance.** Transactions should not be submitted when gas exceeds the configured limits.

Test coverage

Cases covered

- Anchor batch initialization, L2 block polling with concurrent fetching, and block aggregation
- Batch commitment to L1 and TEE/ZK proof requests and commitments
- Batch/proof not-ready scenarios and idempotency for already committed batches
- Window-based processing, economic strategy throttling, and block-trace time-outs

Cases not covered

- Concurrent relayer instances committing same batch
- L2 chain reorganization affecting already processed blocks
- Blob data availability after EIP-4844 pruning period

Attack surface

- **L2 block data from L2 client.** A compromised or malicious L2 RPC endpoint could provide false block traces with incorrect transactions, state roots, or block hashes, potentially causing the relayer to aggregate and commit invalid batches to L1.
- **Proof data from the prover service.** A malicious or compromised prover service could provide invalid cryptographic proofs (TEE attestations or ZK proofs) that do not correctly verify the L2 state transitions, potentially allowing invalid state to be finalized on L1 if accepted.
- **Configuration parameters.** Malicious modification of window length, polling size, or proof types could cause a denial of service (excessively large windows), skip necessary proof types, or alter batch-processing behavior to disrupt rollup operations.
- **L1 contract state queries.** If the L1 RPC endpoint is compromised, queries for `lastCommittedBatch` or `lastVerifiedBatch` could return manipulated values, causing the relayer to skip batches, commit out of order, or double-commit batches.

5.2. Component: L1 Listen Service (IL1ListenService)

Description

The L1 Listen Service polls L1 blocks to extract cross-layer messages and L1 block-fee information for oracle updates. It scans L1 blocks from the last processed height to the latest finalized block (default policy: `FINALIZED`), retrieves L1 → L2 mailbox messages in batches (default 32 blocks per poll), saves messages to database for mailbox service processing, and captures L1 block-fee information (`baseFeePerGas`, `gasUsed`, `excessBlobGas`) for oracle gas-price updates.

The implementation queries the L1 client for the latest block header using a configurable block polling policy, uses a flowable pattern to stream L1 message batches from mailbox-contract events, processes blocks atomically in transactions (saving messages and creating oracle requests), and updates processed L1 block number tracking. When processing the latest block in a poll window, it

creates oracle request records for L1 block-fee updates that the Oracle Service will process.

Invariants

- **Sequential block processing.** L1 blocks must be processed in order without gaps.
- **Start-height initialization.** The start height must be set before polling begins.
- **Block-number monotonicity.** A processed L1 block number always increases.
- **Oracle-request uniqueness.** There is one oracle request per L1 block for fee updates.
- **Message batch atomicity.** All messages in a block batch are saved atomically with a block number update.

Test coverage

Cases covered

- L1 block polling, message batch processing, and L1 block-fee information capture
- Empty message batch handling, block number updates, and oracle-request creation

Cases not covered

- L1 chain reorganizations affecting processed blocks
- RPC failures during block polling
- Message-extraction errors from the mailbox contract

Attack surface

- **L1 block data from RPC.** A compromised L1 RPC endpoint could return false block headers with manipulated gas-fee information (`baseFeePerGas`, `excessBlobGas`) that gets fed to the oracle, causing incorrect L2 gas pricing and potential economic attacks on L2 users.
- **L1 mailbox contract events.** A compromised RPC could inject false mailbox message events that do not exist on chain or omit legitimate messages, causing incorrect message processing on L2 or blocking legitimate cross-layer communications.
- **Configuration.** Manipulation of `maxPollingBlockSize` could cause an excessive RPC load or missed blocks, while changing `blockPollingPolicy` from `FINALIZED` to `LATEST` could expose the relayer to reorganization-based attacks where L1 data is later invalidated.

5.3. Component: Mailbox Service (IMailboxService)

Description

The Mailbox Service bridges communication between L1 and L2, handling both L1 → L2 message delivery and L2 → L1 message proof generation. It scans L1 blocks for mailbox messages and submits them as special transactions to the L2 mailbox contract, tracks message nonces and state transitions (READY → COMMITTED), and generates Merkle proofs for batches of L2 → L1 messages (for withdrawals). The service implements message queue management with pending limits (default 256) and detects/processes missed L1 messages by identifying nonce gaps.

The implementation queries the L2 mailbox contract for current pending and latest nonces, retrieves ready messages from the database in batches (default 32), submits them as L1 message transactions to L2, and creates reliable transaction records for tracking. For L2 → L1 proofs, it uses L2MerkleTreeAggregator to generate proofs per batch after finalization, maintaining a next batch index pointer. The service checks batch finalization before generating proofs to ensure stability.

Invariants

- **Message nonce sequencing.** L1 messages must be processed in strict nonce order.
- **No nonce gaps.** If a gap is detected ($\text{minNonce} > \text{pendingNonce} + 1$), missing messages must be processed first.
- **Pending-message limit.** The number of pending L1 messages on L2 cannot exceed `maxPendingL1Msg` (default 256).
- **Proof batch alignment.** L2 message proofs are generated per batch — must wait for batch finalization.
- **Message state progression.** Messages transition READY → COMMITTED (cannot skip states).
- **Atomic message plus transaction insertion.** L1 message state update and reliable transaction insertion must be atomic.
- **L1 start-height initialization.** Can only be set once.

Test coverage

Cases covered

- Mailbox initialization and L1 message processing and submission to L2
- Detection of packaged/pending messages and nonce gap handling
- L2 message Merkle proof generation and empty proof results
- Pending message limits and state transitions (READY → COMMITTED)

Cases not covered

- Concurrent message processing from multiple relayers
- L1 chain reorganization affecting message ordering

- L2 chain reorganization affecting message nonces

Attack surface

- **L1 message data from the L1 mailbox contract.** Attackers can send malicious or malformed messages from L1 through the mailbox contract (e.g., messages with excessive calldata, reentrant calls, or targeting vulnerable L2 contracts) that the relayer will reliably relay to L2 for execution.
- **L2 message data from block traces.** A compromised L2 RPC endpoint could report false L2 → L1 messages (withdrawals, cross-chain calls) that do not actually exist in finalized blocks, causing the relayer to generate invalid Merkle proofs that would fail verification on L1.
- **L2 mailbox contract state queries.** Manipulated pendingNonce or latestNonce values from a compromised RPC could cause the relayer to skip messages, reprocess messages, or violate nonce ordering, breaking the message-sequencing guarantees.
- **Configuration.** Setting maxPendingL1Msg too low could cause a denial of service to message processing, while setting it too high could overwhelm L2. Manipulating l1MsgNumberPerBatchLimit could alter the message-processing rate and batch sizes.

5.4. Component: Reliable Transaction Service (IReliableTxService)

Description

The Reliable Transaction Service ensures all L1 and L2 transactions submitted by the relayer eventually confirm on chain through retry logic, gas-price escalation, and state tracking. It manages the complete transaction life cycle from submission through final confirmation or failure, preventing transaction loss. The service monitors pending transactions, detects stuck/timed-out transactions, and speeds them up by resubmitting with higher gas prices. It tracks state transitions (PENDING → PACKAGED → SUCCESS / FAILED) and retries failed transactions up to a configurable limit while respecting economic constraints.

The implementation periodically polls the database for nonfinalized transactions, queries blockchain nodes for receipt status, and uses nonce-based transaction replacement for speed-ups (default 600s time-out). It maintains raw transaction data for resubmission with the same nonce and uses Spring transaction management for atomic database updates. The service respects economic strategy constraints by skipping transactions when gas prices exceed thresholds.

Invariants

- **Transaction ordering must be maintained.** Batch commit transactions for batch N+1 cannot be confirmed before batch N.
- **Nonce monotonicity.** Each sender account's nonce must increment sequentially without gaps.
- **At-least-once delivery.** Every initiated transaction must eventually either succeed or be

marked as permanently failed after the retry limit.

- **State-transition consistency.** Once a transaction moves to TX_SUCCESS, the corresponding batch/proof state must be updated atomically.
- **Gas-price-escalation monotonicity.** Replacement transactions must always have a higher gas price than the original.
- **No double-spending.** The same nonce cannot be used for different transaction data (enforced by maintaining raw transaction data).

Test coverage

Cases covered

- L1/L2 pending transaction processing and confirmation
- Time-out transaction speed-ups and failed transaction retries
- Missing nonce detection and oracle-gas-feed-request creation
- Contract warnings, metrics recording, and proof-commit notifications

Cases not covered

- Concurrent transaction submissions from multiple relayer instances for the same batch
- L1/L2 chain reorganization scenarios
- Nonce-collision edge cases with external transactions from the same account

Attack surface

- **Gas prices from the blockchain.** Attackers could artificially inflate L1 gas prices through market manipulation or MEV strategies, causing the relayer's economic strategy to halt transaction submissions and stall the entire rollup until gas prices decrease.
- **Transaction receipts from blockchain nodes.** A compromised RPC endpoint could report false transaction receipts (e.g., marking failed transactions as successful or vice versa), causing the relayer to incorrectly update batch/proof states or skip necessary retries, potentially corrupting the rollup state.
- **Database state.** Direct database access would allow attackers to manipulate reliable transaction records, alter nonces, change transaction states, modify raw transaction data, or delete retry tracking, causing transaction loss or state corruption.
- **Configuration values.** Manipulating retry-limit could exhaust retries prematurely, tx-timeout-limit could prevent timely speed-ups, and process-batch-size could cause performance issues or missed transactions during high-volume periods.

5.5. Component: Oracle Service (IOracleService)

Description

The Oracle Service feeds L1 gas pricing information to L2, enabling accurate L2 transaction cost estimation. It processes two types of fee updates: L1 block-fee updates (base fee and blob base fee using EIP-1559/EIP-4844 formulas) and L2 batch cost updates (DA fee and execution fee from actual batch/proof transaction costs). The service calculates fee-scaling factors and submits them to the L2 gas-oracle contract when changes exceed thresholds. It manages oracle-request states (INIT → COMMITTED / SKIP) and applies fee calibration with upper bounds (max 1 Gwei) and lower bounds (minimum 1% of previous fee).

On start-up, the service initializes from the L2 gas-oracle contract state. It pulls oracle requests in the INIT state from the database (default 10 per round), calculates the next-block fees using EIP-1559 and EIP-4844 formulas, compares against thresholds (default 0%), and submits `updateBaseFeeScala` or `updateBatchRollupFee` transactions to L2. For batch costs, it parses L1 transaction receipts to extract gas prices and usage. The state is maintained in-memory (`latestOracleFeeInfo`) with database synchronization.

Invariants

- **Fee monotonicity with calibration.** Fee updates respect configurable thresholds but are calibrated to prevent extreme values.
- **Batch-fee sequencing.** Batch N's fee update can only be processed after batch N-1's fee update committed.
- **Base-fee-update ordering.** It only processes the highest block number update and skips older ones.
- **Fee-scala bounds.** Base-fee scala must be between 1 and a reasonable upper limit (calibrated to max 1 Gwei).
- **Atomic fee updates.** Fee-scala update and reliable transaction insertion must be atomic.
- **Oracle-request idempotency.** The same oracle request should not be processed twice.
- **Fee-calculation correctness.** It uses proper EIP-1559 and EIP-4844 formulas.

Test coverage

Cases covered

- Service initialization and L1 block and L2 batch-fee processing
- EIP-1559/EIP-4844 fee predictions and fee scala calculations with calibration
- Threshold-based update skipping and outdated request handling
- Batch commit-/prove-request coordination, vault withdrawals, and profit/scala updates

Cases not covered

- Concurrent oracle updates from multiple relayers
- Fee-calculation overflow scenarios

Attack surface

- **L1 transaction receipts.** A compromised RPC could report false gas costs for batch commit and proof transactions (manipulated `gasUsed`, `effectiveGasPrice`, `blobGasPrice`, and `blobGasUsed`), causing the oracle to calculate incorrect batch fees and mispricing L2 transactions.
- **L1 block-fee info.** While `baseFeePerGas`, `gasUsed`, and `excessBlobGas` are consensus-verified block fields, a compromised RPC could return values from different blocks or fabricated data, causing the oracle to predict incorrect next-block fees and update L2 with incorrect gas-price scalas.
- **L2 gas-oracle contract state.** Manipulated queries for `lastBatchDaFee`, `lastBatchExecFee`, or `lastBatchByteLength` from a compromised L2 RPC could cause initialization with incorrect baseline values, leading to cumulative errors in all subsequent fee calculations.
- **Configuration.** Setting `oracleBaseFeeUpdateThreshold` to extreme values (0 = excessive updates/gas waste, 100 = never updates/stale prices) or manipulating fee-calibration bounds could cause L2 users to overpay or underpay for transactions.

6. Assessment Results

During our assessment on the scoped Jovay Relayer targets, we discovered 11 findings. No critical issues were found. Two findings were of high impact, two were of medium impact, five were of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

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

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

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