OpenZeppelin | security

MerkleDB Audit

Ava Labs.

March 13, 2024

Table of Contents

Summary

Type	Infrastructure
Timeline	From 2024-01-08 To 2024-02-16
Languages	Golang

Total Issues	9 (8 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	3 (3 resolved)
Notes & Additional Information	5 (4 resolved)

Scope

We audited the x/merkledb and x/sync packages from the ava-labs/avalanchego repo at commit 73c4c0f.

In scope were the following files:

```
x
├── merkledb
│   ├── batch.go
│   ├── cache.go
│   ├── codec.go
│   ├── db.go
│   ├── history.go
│   ├── intermediate_node_db.go
│   ├── key.go
│   ├── metrics.go
│   ├── node.go
│   ├── proof.go
│   ├── tracer.go
│   ├── trie.go
│   ├── value_node_db.go
│   ├── view.go
│   └── view_iterator.go
└── sync
    ├── client.go
    ├── db.go
    ├── manager.go
    ├── metrics.go
    ├── network_client.go
    ├── network_server.go
    ├── response_handler.go
    ├── workheap.go
    └── g_db
        ├── db_client.go
        └── db_server.go
```

System Overview

A MerkleDB is an implementation of a persisted key-value store using a Merkle radix trie, which is a structure that combines a Merkle tree and a radix trie. It was designed to provide a secure, efficient, and verifiable key-value store that leverages the strengths of Merkle trees and radix tries. It is particularly useful for applications requiring high security and verifiability, such as blockchain technologies.

The fundamental advantages of this database come from its efficient storage and retrieval of key-value pairs, which allows for the validation of data integrity and the ability to verify whether specific key-value pairs are present or absent within the database. The usage of a Merkle tree enables the database to create a fingerprint (hash) of the entire dataset by hashing the data at the leaves and then recursively hashing the concatenation of child hashes until a single hash (known as the root hash) represents the entire dataset. This root hash serves as a unique identifier for the data state at any given time, as any modification of its underlying key-values would result in a different root hash.

This MerkleDB implementation also introduces the concept of views, which are essentially proposals for database operations. A view is created out of a set of key-value operations (insert, update, delete). Once created, views are immutable and can be chained together, allowing for complex data manipulation and versioning before any changes are permanently committed to the database.

The database supports the generation of three types of cryptographic proofs to ensure the integrity and verifiability of data within its key-value store: simple proofs, range proofs, and change proofs.

- **Simple Proofs:** Used to demonstrate that a specific key-value pair either exists or not within the database at a certain state (which, as mentioned before, is identified by a specific root hash). These proofs rely on a path of hashes from the target node up to the root of the tree. For inclusion proofs, the path ends at the node containing the key. For exclusion proofs, it ends at a node where the key would be located if it were present, providing evidence of the key's absence.

- **Range Proofs:** An extension of simple proofs to cover a contiguous set of key-value pairs within a specified key range. They prove that all key-value pairs within the range are present in the database at a particular state and that no additional keys exist within this

range. These proofs are particularly useful for efficiently verifying large portions of the database, allowing clients to synchronize or download multiple key-value pairs in a single operation.

- **Change Proofs:** Show the differences in data between two states of the database, identified by their respective root hashes. These proofs contain the set of key-value pairs that have been added, modified, or deleted between the two states. Change proofs enable clients to update their local copy of the database to a new state by applying the changes indicated in the proof, without needing to reverify the entire database.

Through the usage of these proofs, the `sync` package allows for the synchronization of a MerkleDB instance between a client and a server, where the server holds an up-to-date version of the database, and the client has an outdated or empty version. This synchronization mechanism is ideal for blockchain systems and is intended to eventually be compatible with Firewood, an upcoming state-of-the-art database optimized for storing Merkleized blockchain state.

Methodology

In order to assess the correctness of the MerkleDB implementation, we focused on the following aspects during the audit:

- Examine node serialization and deserialization to ensure that the process is secure against malformed or malicious data.
- Verifying node hashing and ID uniqueness to ensure that it reliably produces unique and collision-resistant node IDs.
- Assess locking mechanisms and concurrency controls to ensure that they provide the necessary atomicity and consistency without leading to deadlocks or performance issues.
- Examine how views are committed to the MerkleDB and how their validity is tracked and managed, particularly in relation to parent and child views.
- Review the generation and verification of proofs including simple, range, and change proofs.
- Assess the robustness of the synchronization between different MerkleDB instances while ensuring data integrity and client availability.

Evaluation

During the audit of the `merkledb` and the `sync` packages, the codebase was found to be of exceptionally high quality. The code was overall clean, adhering to the best Golang practices and had comprehensive documentation, unit testing, extensive fuzzing, and excellent testing strategies that collectively ensured the correct implementation of the database. However, it is important to note that despite the high quality of the MerkleDB implementation, it has yet to undergo extensive real-world testing. Given the complexities and potential unforeseen failures inherent in distributed systems, it is highly likely that a code audit alongside traditional testing methods is not sufficient to uncover all existing issues.

Therefore, it is highly advisable to develop simulation software that can meticulously test the database and its sync protocol under a wide array of conditions, thereby ensuring its robustness and reliability in real-world scenarios. Remarkably, the AVA Labs team has made significant progress in this regard as well, as during the audit, they provided access to a codebase specifically designed for testing the MerkleDB and its sync protocol.

Nevertheless, we suggest broadening the scope of the simulation software to encompass a more diverse array of testing scenarios such as hardware failures, fluctuating network latencies, and data corruption scenarios to ensure MerkleDB robustness and reliability in diverse and challenging environments.

High Severity

H-01 Ability to Generate Valid Exclusion Proofs of Keys Which Are Present in the Trie

The MerkleDB allows for the creation of inclusion and exclusion proofs. An inclusion proof verifies the presence of a particular key-value pair in the trie under a specified root, whereas an exclusion proof confirms the absence of a particular key-value pair from the trie under the same root. Both proofs are constructed in a similar manner utilizing the `Proof` struct. For exclusion proofs, this structure's `Key` field is assigned the key intended to be demonstrated as absent from the trie, and the `Value` field is left blank. The `Path` represents a sequence of nodes leading from the trie's root to (the parent of) the position where the node would have been located if it were part of the trie.

During verification of the exclusion proof, the verifier constructs an empty trie and populates it with all the nodes in the `Path`. If the root ID of this constructed trie matches the expected root ID, the verifier can be sure that all the nodes in the `Path` are correct, confirming the absence of the specified key-value pair from the trie.

However, a malicious prover could forge a valid exclusion proof for a key-value pair that actually exists in the trie. This is done by removing the `Value` from the proof and reducing the length of the proof path by not including the parent of the node where the key-value pair would have been located. Given that the ID represents the hash of a node, and each node keeps the IDs of its children, the proof path does not actually have to reach this node as long as one of its parents is included in the proof path. Because of this, the trie created by the verifier will be identical to the upper half of the original trie and yield an identical root ID, and mistakenly pass the verification process.

Consider reviewing the proof verification mechanism to prevent the acceptance of falsified exclusion proofs for existing key-value pairs.

Update: *Resolved in pull request #2789.*

Low Severity

L-01 `RecordKeyChange` Ignores `ErrorNotFound`

When the `recordKeyChange` function in `view.go` gets called with the boolean parameter `newNode` set to `false`, the function calls `v.getParentTrie().getEditableNode(key, hadValue)`. However, if there is an error of type `database.ErrNotFound`, the error gets ignored.

If this edge case happens, the resulting `before` value from calling `v.getParentTrie().getEditableNode(key, hadValue)` would be `nil`, which would be registered in the `v.changes.nodes` mapping. This would be inconsistent as it should not be possible that there is no `before` if this is not a new node. So while in theory, this error should never happen as long as the caller of `recordKeyChange` is properly using the `newNode` parameter, the function should nonetheless fail in case the caller misuses it.

Consider not ignoring the return error `database.ErrNotFound` and having the function return an error.

Update: *Resolved in pull request #2743.*

L-02 If There Is No Existing Child, the Function `addPathInfo` Will Use the Previous `compressedKey`

In the `addPathInfo` function, the verifier adds each node in the proof path to a new trie. For each node, the children are added if they are smaller than the lower bound or greater than the upper bound of the keys which are proven. While adding the children, the function will fetch the corresponding `compressedKey` before creating the child entry.

However, because the `compressedKey` variable is declared outside the `for` loop, if there is no existing child in the constructing trie, the function `n.setChildEntry` will use the `compressedKey` from the last child that was found instead of passing an empty `Key`. This results in a trie with nodes storing the wrong `compressedKey` for their children. However, as only the ID is necessary for calculating the hash, this should not affect the proof verification process.

Consider if setting the `compressedKey` is absolutely necessary in the `addPathInfo` function. If it is, then ensure that it has the correct value for each child.

Update: *Resolved in pull request #2777.*

L-03 Incorrect Parameter Value for the `recordKeyChange` Function When Deleting a Node

The `recordKeyChange` function has a `hadValue` boolean parameter that is passed when calling the function `getEditableNode`. This is necessary to select which underlying database (`valueNodeDB` or `intermediateNodeDB`) should be used when looking up the node in the underlying database.

However in the `view.remove` function, the node value is set to `maybe.Nothing` and then `recordNodeDeleted` is called which calls `recordKeyChange`. So in this case, the `hadValue` would be set to `false` when the deleted node did have an existing value previously. This would result in not getting the previous value from the parent trie because it will be selecting the wrong database.

Consider reviewing how the `hadValue` parameter is being set instead of always using `after.hasValue()` to ensure that it is properly identifying whether a node did have a previous value.

Update: *Resolved in pull request #2779.*

Notes & Additional Information

N-01 Possible Optimization on Trie Building When Verifying a Proof

When a proof is verified, the function `addPathInfo` iterates from the last element `len(proofPath) - 1` in the proof path and then proceeds backwards to the first element. This reverse order indicates that the trie is built by processing nodes starting from the leaves

and moving upwards towards the root in a bottom-to-top approach. However, it is possible that building the trie from a top-to-bottom approach would be more efficient, as it would minimize the number of times that a node has to be created and then deleted to be replaced with a split.

Consider evaluating whether a top-to-bottom approach when processing the nodes in `addPathInfo` would be more efficient.

Update: *Acknowledged, not resolved.*

N-02 `ViewNodeCacheMiss` Is Never Used

The `ViewNodeCacheHit` function is called when a node is successfully retrieved from the cache. However, the `ViewNodeCacheMiss` function is never used.

Consider calling the `ViewNodeCacheMiss` function when a node fails to be retrieved from the cache.

Update: *Resolved in pull request #2781 and pull request #2844.*

N-03 Code Style Inconsistencies

Throughout the codebase, there are several places where code has an inconsistent style:

- In the assignment of `provenKey`, the `lastNode` variable could be reused.
- The range instruction could iterate over both the key-value pairs instead of only the key. This way, `childEntry` should not be declared separately.

Consider reviewing the entire codebase to improve consistency and refactor the code where possible.

Update: *Resolved in pull request #2783.*

N-04 Misleading Comments

The following misleading and inconsistent comments have been identified in the codebase:

- According to this comment, `visitPathToKey` should return the nodes along the path. Instead, the `visitNode` function is called on each node along the path and `visitPathToKey` either returns `err` or `nil`.

- The comment in `getValue` refers to a local "copy of the key", but it should refer to a "copy of the value at the key".
- According to this comment, `compressNodePath` is merging nodes without a value and a single child recursively. However, it is only merging one single node with its parent.
- The documentation of `getProof` mentions a non-existent `bytesPath` variable.
- This comment in `verifyProofPath` says "should store a value". However, it should be "should not store a value".

Consider revising the comments to improve consistency and more accurately reflect the implemented logic.

Update: *Resolved in pull request #2780.*

N-05 Variable Shadowing

Through the codebase, there are multiple instances of variable shadowing where a variable takes the name of an existing function:

- The `newView` variable in `NewView` is shadowing the newView function.
- The `newView` variable in `newView` is shadowing the newView function.
- The `newView` variable in `newViewWithChanges` is shadowing the newView function.

While it does not affect the code as the shadowing only happens in the scope of the function, nonetheless, consider following the best practices and using a different variable name.

Update: *Resolved in pull request #2784.*

Conclusion

The MerkleDB is a key-value database that inherits the properties of Merkle trees and radix tries to offer an innovative solution for secure and efficient data storage and verification. This combination enables the database to efficiently manage data while ensuring its integrity through cryptographic proofs, making it particularly suitable for blockchain applications where security and data verifiability are paramount.

Throughout the audit, a high-severity issue was identified, in addition to several issues of lesser severity. Despite this, the codebase was found to be of very high quality, having extensive documentation and a comprehensive suite of tests that covered a large surface of the codebase.

However, because of the complexities inherent in distributed systems, we strongly recommend the development of simulation software for testing and ensuring the reliability and robustness of the database in a variety of conditions. This simulation software should allow for testing the operation of the MerkleDB in diverse and challenging environments, ensuring that MerkleDB can meet the demands of real-world scenarios.