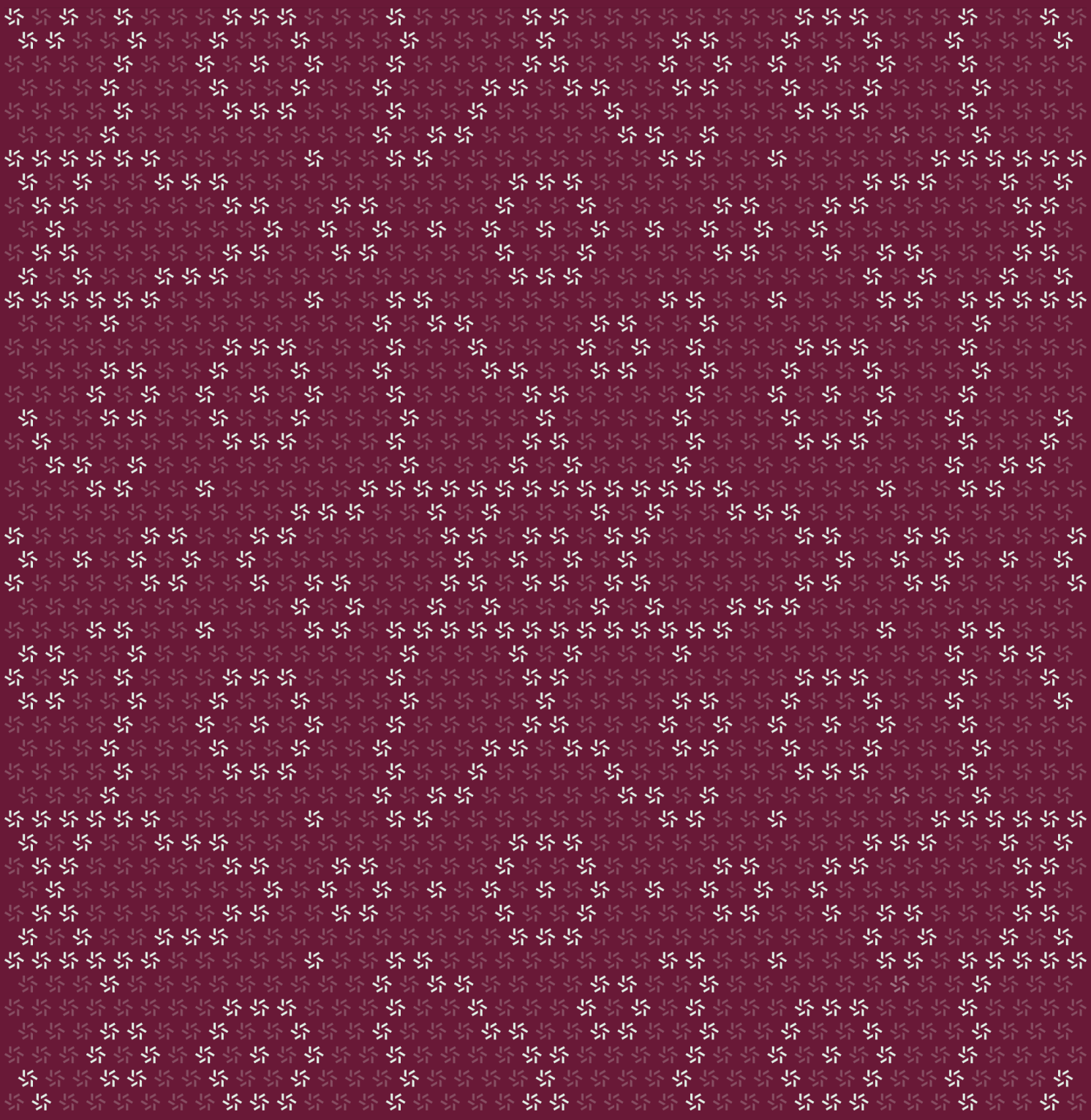
Zellic

Prepared for
Larry Lyu
Left Curve Software Ltd.

Prepared by
Gunhee Ahn
Avraham Weinstock
Zellic

October 22, 2024

Grug
Blockchain Security Assessment



Contents

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 Left Curve Software Ltd. from October 11th to October 18th, 2024. During this engagement, Zellic reviewed Grug'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:

- Can an attacker generate a malicious proof?
- Can an attacker manipulate the state stored on the blockchain?
- Are there any vectors that can trigger a denial of service?
- Are all functions, including membership and nonmembership proofs, working properly without any issues?

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. Additionally, as Grug is still a work in progress at the time of writing, it is recommended to continue adding unit and property tests with new features, and to have new features audited when they are complete.

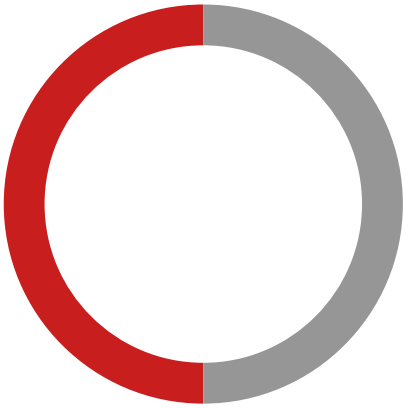
1.4. Results

During our assessment on the scoped Grug crates, we discovered two findings. One critical issue was found. The other finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Left Curve Software Ltd. in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	0
Low	0
Informational	1



2. Introduction

2.1. About Grug

Left Curve Software Ltd. contributed the following description of Grug:

> Grug is an execution environment for blockchains. The scope of this audit is the state commitment scheme that Grug uses, the Jellyfish Merkle Tree (JMT).

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 crates.

Nondeterminism. Nondeterminism is a leading class of security issues on Cosmos. It can lead to consensus failure and blockchain halts. This includes but is not limited to vectors like wall-clock times, map iteration, and other sources of undefined behavior (UB) in Go.

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

Complex integration risks. Several high-profile exploits have been the result of unintended consequences when interacting with the broader ecosystem, such as via IBC (Inter-Blockchain Communication Protocol). Zellic will review the project's potential external interactions and summarize the associated risks. If applicable, we will also examine any IBC interactions against the ICS Specification Standard to look for inconsistencies, flaws, and vulnerabilities.

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 crates itself. These observations — found in the Discussion (4. ↗) 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:

Grug Crates

Type	Rust
Platform	Cosmos
Target	left-curve
Repository	https://github.com/left-curve/left-curve ↗
Version	`64e311bcb5c0df657be533432e02ed6a0371ef08`
Programs	`grug/jellyfish-merkle` `grug/db-disk`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.8 person-weeks. The assessment was conducted by two consultants over the course of 1.2 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Gunhee Ahn
Engineer
gunhee@zellic.io

Avraham Weinstock
Engineer
avi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 11, 2024	Kick-off call
October 11, 2024	Start of primary review period
October 18, 2024	End of primary review period

3. Detailed Findings

3.1. Reachable `unwrap` panic in `ics23_prove`

Target	grug/db-disk/src/db.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `ics23_prove` function attempts to generate an ICS-23 proof of membership or nonmembership of a given key in Grug's state based on whether or not the key is present or absent.

It checks if there is a corresponding `value` for the key in the `state_storage` RocksDB table, and if there is, it returns a proof of membership. However, if there is no corresponding value, it will attempt to generate a nonmembership proof, which consists of membership proofs for neighboring keys that would be adjacent to the given key if it were present.

To find these neighboring keys, it attempts to query the `preimages` RocksDB table (which contains mappings from hashes of keys to keys) to find the next and previous keys to the queried key ordered by hash value. It then obtains the value of those keys in the `state_storage` table, assuming they exist, and generates membership proofs for them.

```
...
    let proof = match state_storage.read(&key) {
        // Value is found. Generate an ICS-23 existence proof.
        Some(value) =>
    CommitmentProofInner::Exist(generate_existence_proof(key, value)?),
        // Value is not found.
        //
        // Here, unlike Diem or Penumbra's implementation, which walks the
        // tree to find the left and right neighbors, we use an approach
        // similar to SeiDB's:
        // https://github.com/sei-protocol/sei-
    db/blob/v0.0.43/sc/memiavl/proof.go#L41-L76
        //
        // We simply look up the state storage to find the left and right
        // neighbors, and generate existence proof of them.
        None => {
            let cf = cf_preimages(&self.inner.db);
            let key_hash = key.hash256();

            let opts = new_read_options(Some(version), None, Some(&key_hash));
            let left = self
```

```
                .inner
                .db
                .iterator_cf_opt(&cf, opts, IteratorMode::End)
                .next()
                .map(|res| {
                    let (_, key) = res?;
                    let value = state_storage.read(&key).unwrap();
                    generate_existence_proof(key.to_vec(), value)
                })
                .transpose()?;

            let opts = new_read_options(Some(version), Some(&key_hash), None);
            let right = self
                .inner
                .db
                .iterator_cf_opt(&cf, opts, IteratorMode::Start)
                .next()
                .map(|res| {
                    let (_, key) = res?;
                    let value = state_storage.read(&key).unwrap();
                    generate_existence_proof(key.to_vec(), value)
                })
                .transpose()?;

            CommitmentProofInner::Nonexist(NonExistenceProof { key, left,
    right })
        },
    };
...
```

However, the `commit` function can delete the keys and values inserted in the `state_storage` and `state_commitment` tables without deleting their corresponding key hashes in the `preimages` table.

```
...
    // Writes in state commitment
    let cf = cf_state_commitment(&self.inner.db);
    for (key, op) in pending.state_commitment {
        if let Op::Insert(value) = op {
            batch.put_cf(&cf, key, value);
        } else {
            batch.delete_cf(&cf, key);
        }
    }

    // Writes in state storage (note: don't forget timestamping)
    let cf = cf_state_storage(&self.inner.db);
```

```
    for (key, op) in pending.state_storage {
        if let Op::Insert(value) = op {
            batch.put_cf_with_ts(&cf, key, ts, value);
        } else {
            batch.delete_cf_with_ts(&cf, key, ts);
        }
    }
...
```

Because of this, the keys can be absent from the `state_storage` table despite the preimages remaining present, causing the `unwrap`s to panic in the process of finding the neighboring nodes.

This is demonstrated by the following test:

```
#[test]
fn ics23_prove_delete() {
    let path = TempDataDir::new("_grug_disk_db_ics23_delete");
    let db = DiskDb::open(&path).unwrap();
    let (_, maybe_root) = db.flush_and_commit(Batch::from([
        (b"m".to_vec(), Op::Insert(b"m".to_vec())), (b"a".to_vec(),
    Op::Insert(b"a".to_vec()))
    ])).unwrap();
    let root = maybe_root.unwrap().to_vec();
    let to_prove = b"L";
    let (_, maybe_root) = db.flush_and_commit(Batch::from([(b"a".to_vec(),
    Op::Delete)])).unwrap();
    let root = maybe_root.unwrap().to_vec();
    let proof = db.ics23_prove(to_prove.to_vec(), None).unwrap();
    println!("{:?}", proof);
    assert!(ics23::verify_non_membership::<HostFunctionsManager>(&proof,
    &ICS23_PROOF_SPEC, &root, to_prove));
}
```

Impact

As nonmembership proofs are requested in the process of handling ICS-4 packet time-outs, this will result in periodic crashes / denial of service whenever the hash of a packet-receipt path is adjacent to a deleted key.

Additionally, the use of `SeekToLast` with an upper bound smaller than the last key is undefined by RocksDB ↗, and this is the current behavior with `set_iterator_upper_bound` (in `new_read_options`) and `IteratorMode::End`.

Recommendations

We recommend addressing the panic by using `.find_map` in place of `.next().map`, with error handling changed to skip over preimages that are no longer present in the state, and addressing the RocksDB undefined behavior by using `IteratorMode::From` and not using lower and upper bounds.

The following patch applies the above suggestions to `ics23_prove`:

```
 let cf = cf_preimages(&self.inner.db);
 let key_hash = key.hash256();

let opts = new_read_options(Some(version), None, Some(&key_hash));
let opts = new_read_options(Some(version), None, None);
 let left = self
     .inner
     .db
    .iterator_cf_opt(&cf, opts, IteratorMode::End)
    .next()
    .map(|res| {
        let (_, key) = res?;
        let value = state_storage.read(&key).unwrap();
        generate_existence_proof(key.to_vec(), value)
    .iterator_cf_opt(&cf, opts, IteratorMode::From(&key_hash, rocksdb::
        Direction::Reverse))
    .find_map(|res| {
        let key = match res {
            Ok((_, key)) => key,
            Err(e) => return Some(Err(DbError::from(e))),
        };
        let value = state_storage.read(&key)?;
        Some(generate_existence_proof(key.to_vec(), value))
    })
    .transpose()?;

let opts = new_read_options(Some(version), Some(&key_hash), None);
let opts = new_read_options(Some(version), None, None);
 let right = self
     .inner
     .db
    .iterator_cf_opt(&cf, opts, IteratorMode::Start)
    .next()
    .map(|res| {
```

```
        let (_, key) = res?;
        let value = state_storage.read(&key).unwrap();
        generate_existence_proof(key.to_vec(), value)
    .iterator_cf_opt(&cf, opts, IteratorMode::From(&key_hash, rocksdb::
        Direction::Forward))
    .find_map(|res| {
        let key = match res {
            Ok((_, key)) => key,
            Err(e) => return Some(Err(DbError::from(e))),
        };
        let value = state_storage.read(&key)?;
        Some(generate_existence_proof(key.to_vec(), value))
    })
    .transpose()?;
```

Additionally, we recommend removing keys from the `preimages` table when they are removed from the `state_storage` table to conserve space and improve the efficiency of iterating to find neighboring nodes.

Remediation

This issue has been acknowledged by Left Curve Software Ltd., and fixes were implemented in the following commits:

- 5fe90299 ↗
- 3d2c627d ↗

3.2. Diem-style proof construction fails for the empty tree

Target	grug/jellyfish-merkle/src/tree.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `MerkleProof::prove` function for constructing Diem-style proofs of nonmembership returns an error when attempting to look up the root node in the empty tree. This is demonstrated by the following test:

```
#[test]
fn test_removal_nonmembership() {
    let mut storage = MockStorage::new();
    TREE.apply_raw(&mut storage, 0, 1, &Batch::from([(b"a".to_vec(),
    Op::Insert(b"a".to_vec()))])).unwrap();
    TREE.apply_raw(&mut storage, 1, 2, &Batch::from([(b"a".to_vec(),
    Op::Delete)])).unwrap();
    let proof = TREE.prove(&storage, b"a".hash256(), 2);
    assert!(matches!(proof, Ok(Proof::NonMembership(_))), "{:?}", proof);
}
```

Impact

As Diem-style proofs are not used for IBC, this does not risk funds being locked by failure to prove an ICS-4 packet time-out. However, Diem-style proofs are constructed through ABCI `"/store"` queries, which will incorrectly return an error instead of successfully returning a nonexistence proof if the store is empty.

Recommendations

Use an internal node with neither child present to represent the empty tree in proof construction and verification instead of permitting the root to be absent.

Remediation

This issue has been acknowledged by Left Curve Software Ltd., and a fix was implemented in commit 58da8b70 ↗.

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. Property testing of `ics23_prove`

The following property test consists of multiple batches of insertions and deletions in a pattern that covers every case of applying operations to a tree, except for those explicitly marked as unreachable. We recommend its inclusion in the test suite to mitigate potential future bugs as the code evolves.

```
proptest! {
    #[test]
    fn proptest_apply_ics23prove(
        (batch1, removals_batch1) in prop::collection::hash_map("[a-z]{1,10}",
    "[a-z]{1,10}", 1..100).prop_flat_map(|batch1| {
            let len = batch1.len();
            (Just(batch1),
    prop::collection::vec(any::<prop::sample::Selector>(), 0..len))
        }),
        batch2 in prop::collection::hash_map("[a-z]{1,10}", "[a-z]{1,10}",
    1..100),
        removals_batch2 in
    prop::collection::vec(any::<prop::sample::Selector>(),
    0..50)
    ) {
        let path = TempDataDir::new("_grug_disk_db_apply_ics23prove");
        let db = DiskDb::open(&path).unwrap();
        use std::collections::BTreeMap;
        let mut state = BTreeMap::new();
        let (_, maybe_root)
    = db.flush_and_commit(Batch::from(batch1.clone().into_iter().map(|(k,
    v)| (k.into_bytes(),
    Op::Insert(v.into_bytes()))).collect::<BTreeMap<_,_>>())).unwrap();
        let root0 = maybe_root.unwrap().to_vec();
        for (k, v) in batch1.iter() {
            state.insert(k.clone().into_bytes(), v.clone().into_bytes());
        }
        for (k, v) in state.iter() {
            let proof = db.ics23_prove(k.clone(), None).unwrap();
            assert!(matches!(proof, ics23::CommitmentProof { proof:
    Some(ics23::commitment_proof::Proof::Exist(_)) }));
            assert!(ics23::verify_membership::<HostFunctionsManager>(&proof,
    &ICS23_PROOF_SPEC, &root0, &k, &v));
        }
```

```
        let (_, maybe_root)
    = db.flush_and_commit(Batch::from(removals_batch1.iter().map(|r|
    (r.select(&batch1).0.clone().into_bytes(),
    Op::Delete)).collect::<BTreeMap<_,_>>())).unwrap();
        let root1 = maybe_root.unwrap().to_vec();
        for r in removals_batch1.iter() {
            let pre_k = r.select(&batch1).0;
            let k = pre_k.clone().into_bytes();
            state.remove(&k);
            let proof = db.ics23_prove(k.clone(), Some(0)).unwrap();
            assert!(matches!(proof, ics23::CommitmentProof { proof:
    Some(ics23::commitment_proof::Proof::Exist(_)) }), "proof: {:?}, k: {:?}",
    proof, pre_k);
            assert!(ics23::verify_membership::<HostFunctionsManager>(&proof,
    &ICS23_PROOF_SPEC, &root0, &k, &batch1[pre_k].clone().into_bytes()));
            let proof = db.ics23_prove(k.clone(), None).unwrap();
            assert!(matches!(proof, ics23::CommitmentProof { proof:
    Some(ics23::commitment_proof::Proof::Nonexist(_)) }));

    assert!(ics23::verify_non_membership::<HostFunctionsManager>(&proof,
    &ICS23_PROOF_SPEC, &root1, &k));
        }
        let batch_batch2 = Batch::from(
            removals_batch2.iter().map(|r|
    (r.select(&batch1).0.clone().into_bytes(),
    Op::Delete))
            .chain(batch2.clone().into_iter().map(|(k, v)| (k.into_bytes(),
    Op::Insert(v.into_bytes()))))
            .collect::<BTreeMap<_,_>>()
        );
        let (_, maybe_root) = db.flush_and_commit(batch_batch2).unwrap();
        let root2 = maybe_root.unwrap().to_vec();
        for r in removals_batch2.iter() {
            let pre_k = r.select(&batch1).0;
            let k = pre_k.clone().into_bytes();
            state.remove(&k);
            if !batch2.contains_key(pre_k) {
                let proof = db.ics23_prove(k.clone(), None).unwrap();
                assert!(matches!(proof, ics23::CommitmentProof { proof:
    Some(ics23::commitment_proof::Proof::Nonexist(_)) }));

    assert!(ics23::verify_non_membership::<HostFunctionsManager>(&proof,
    &ICS23_PROOF_SPEC, &root2, &k));
            }
        }
        for (k, v) in batch2.iter() {
            state.insert(k.clone().into_bytes(), v.clone().into_bytes());
```

```
        }
        for (k, v) in state.iter() {
            let proof = db.ics23_prove(k.clone(), None).unwrap();
            assert!(matches!(proof, ics23::CommitmentProof { proof:
    Some(ics23::commitment_proof::Proof::Exist(_)) }));
            assert!(ics23::verify_membership::<HostFunctionsManager>(&proof,
    &ICS23_PROOF_SPEC, &root2, &k, &v));
        }
    }
}
```

This recommendation has been acknowledged by Left Curve Software Ltd., and a similar test was incorporated in commit `148fa0a6`↗.

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. Key operations

The Merkle tree proof and its state storage in Grug are primarily managed by the code under `grug/src/jellyfish-merkle/src` and `grug/src/db-disk/src`.

The key operations within our audit scope are as follows:

1. **Applying batches of operations to the Merkle tree.** These functions apply batches of operations produced by block processing, which consist of key-value pairs to insert or delete, to the Merkle tree. Applying batches of operations to the database is done in the ABCI FinalizeBlock handler, and they are committed to the database in the ABCI Commit handler.
2. **ICS-23 proof generation.** These functions implement generation of membership and nonmembership proofs for Grug's state in the ICS-23 format used by IBC. ICS-23 proofs are used for connection, channel, and packet management in the ICS-2, ICS-3, and ICS-4 standards.

Applying batches of operations to the Merkle tree

The application of batch operations resulting from block processing is performed through the `flush_but_not_commit` function, which calls `MERKLE_TREE.apply_raw(...)`. This function takes four parameters — `storage`, `old_version`, `new_version`, and `batch` — and invokes the `MerkleTree::apply` function. The `apply` function takes the same parameters as `apply_raw`, except the keys in the batch must have been hashed and sorted by the caller.

The `apply` function marks the root node of `old_version` as orphaned if it exists in the nodes, starting from `new_version`. Then, the `MerkleTree::apply_at` function is used to apply the appropriate batch operation based on the state of the node at the bits position.

- If it is a leaf node, the `MerkleTree::apply_at_leaf` function is called, which applies a single insertion/deletion in place, or it creates a subtree if there are multiple insertions/deletions to apply to the leaf's position.
- If it is an internal node, the `MerkleTree::apply_at_internal` function is called, which partitions the batch into operations that apply to the left/right subtree, and it recurses into those by calling `MerkleTree::apply_at_child`.
- If the node does not exist, the `MerkleTree::create_subtree` function is used to create a subtree.

After each function completes its assigned role, the state of the node after the operation is saved, and a flush is performed to record this in the `state_commitment` function.

ICS-23 proof generation

The `DiskDb::ics23_prove` function generates proofs that key-value pairs are present or absent from Grug's state. It is intended to be called from IBC packet–processing code, which does not yet appear to be implemented. This function takes two parameters: `key` and `version`. It verifies whether the version is valid (i.e., not greater than the latest version or smaller than the oldest version) and retrieves the appropriate state storage for the given version.

If the state storage contains a `value` corresponding to the `key`, membership proof is performed using the `key` via a call to `MerkleTree::ics23_prove_existence`. If no value is found, a nonmembership proof is conducted by searching neighboring nodes of the nonexistent key and generating membership proofs for those neighboring nodes. This currently does not correctly account for neighbors that may have been deleted; see Finding 3.1. ↗.

To perform membership proofs, `MerkleTree::ics23_prove_existence` traverses child nodes based on the hash of the node's key: if the bit at the current traversal index is 0, the left node is traversed; if it is 1, the right node is traversed. This traversal method is similar to that of a Patricia Merkle tree.

5.2. Security considerations

As the ICS-23 reference implementation is used to verify proofs generated by this process, and since the keys are sorted by their hashes when being stored in the tree, soundness of the proofs is provided by the upstream implementation (i.e., chains cannot equivocate between membership and nonmembership of the same key for a given root hash).

Completeness issues arise from incorrect handling of deletions (as in Finding 3.1. ↗).

As discussed in Finding 3.2. ↗, the `MerkleProof::prove` function, which is used for Diem-style proofs, fails to perform correct verification when attempting to query the root node in an empty tree, also resulting in incompleteness for that proof format.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed.

During our assessment on the scoped Grug crates, we discovered two findings. One critical issue was found. The other finding was 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.