Scroll zkTrie

Security Assessment

September 8, 2023

Prepared for:
Haichen Shen
Scroll

Prepared by: **Filipe Casal, Joe Doyle, and Sanketh Menda**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Scroll engaged Trail of Bits to review the security of zkTrie. The project is a Go implementation of a Merkle-Patricia tree, along with Rust bindings.

A team of three consultants conducted the review from June 26 to July 11, 2023, for a total of four engineer-weeks of effort. Our testing efforts focused on checking for inclusion and non-inclusion proof verification, data structure correctness, and safe and reliable bindings. With full access to the source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

We identified several high-severity findings that undermine the assumptions of the Merkle tree data structure. Flaws in domain separation and the way that the nodes and leaves are hashed allow an attacker to convince a verifier that a key is simultaneously present in the tree and not present in the tree (TOB-ZKTRIE-1). The hashing scheme also allows an attacker to easily create different leaves with the same hash (TOB-ZKTRIE-5) and leaves with different "canonical values" but the same hash (TOB-ZKTRIE-7). An unchecked cast in the Rust bindings also allows an attacker to generate nodes from different byte arrays that would have the same hash (TOB-ZKTRIE-19).

We found several denial-of-service attack vulnerabilities that would allow an attacker to crash the proof verifier with malformed proofs (TOB-ZKTRIE-2) and to cause crashes from the Rust bindings by triggering a mishandling of Cgo handles (TOB-ZKTRIE-12) and by providing maliciously chosen data arrays (TOB-ZKTRIE-19).

We found two cases of vulnerable unsafe Rust code: a data race that can be triggered through a safe Rust interface (TOB-ZKTRIE-16) and an out-of-bounds read (TOB-ZKTRIE-18).

We also found several general software quality problems. In addition to the code quality findings listed in appendix C, we found duplicated functionality (TOB-ZKTRIE-3), an empty function implementation (TOB-ZKTRIE-6), and an unused argument in a core function (TOB-ZKTRIE-9).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Scroll take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Clearly define the library interface and clean up the current implementation.** The Rust and Go APIs should be fully documented and should match where appropriate. Redundant functionality should be removed.

- **Explicitly specify the security guarantees and requirements of the Merkle tree.** The Merkle proof format and verification algorithm should be explicitly specified. If possible, security proofs should be written to confirm that Merkle proof verification is sound based only on the collision resistance of the underlying hash function.

- **Add extensive data structure and API testing.** All functions should have associated positive and negative tests, covering all return paths. Publicly exposed functions should also be fuzz tested. Add static analysis tools like golangci-lint and Semgrep to the CI/CD pipeline.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	5
Medium	3
Low	2
Informational	9

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	2
Cryptography	3
Data Exposure	1
Data Validation	8
Error Reporting	1
Testing	1
Undefined Behavior	3

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineers were associated with this project:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Sanketh Menda, Consultant
sanketh.menda@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 26, 2023	Pre-project kickoff call
June 30, 2023	Status update meeting #1
July 13, 2023	Delivery of report draft
July 13, 2023	Report readout meeting
September 8, 2023	Delivery of comprehensive report
October 10, 2023	Delivery of comprehensive report with fix review appendix

Project Goals

The engagement was scoped to provide a security assessment of the Scroll zkTrie. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the codebase follow best practices for Go and Rust?
- Is the zkTrie implementation sound and complete? That is, does the implementation correctly reject invalid inclusion and non-inclusion proofs in the Merkle tree and correctly accept valid proofs?
- Does the zkTrie implementation correctly handle malformed or malicious inputs?
- Are the Rust bindings to the Go implementation safe? Do the safe-Rust interfaces provide the guarantees required by the Rust type system?

Project Targets

The engagement involved a review and testing of the following target.

zktrie

Repository	github.com/scroll-tech/zktrie
Version	`90179c19281670f41c54bd80ab01e4d64c860521`
Types	Go, Rust
Platform	Native

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Tree implementation:** We manually reviewed the implementation of the Merkle tree, focusing on inclusion and non-inclusion proof generation and verification as well as the hash function usage and correct domain separation. We used test coverage tools to identify areas of code that are not currently being tested; used several static analysis tools to identify potential vulnerabilities; and wrote several tests, including a fuzz test for the proof verification routine and a functional randomized test for the Merkle tree functionality.
- **`lib.go` and `lib.rs`:** We manually reviewed the Go implementation, including functions exposed to C through `cgo`, taking into consideration the correct handling of the `cgo` handles. We manually reviewed the Rust bindings, focusing on unsafe code, correct data handling from publicly exposed functions, and their interaction with the Go library. We used several static analysis tools to identify potential vulnerabilities and wrote proof-of-concept tests for the findings in this report. These tests are included in this report and should be incorporated into Scroll's testing suite.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
golangci-lint	An aggregator of Go linters	Appendix D.1
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix D.2
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix D.3
Go cover and cargo llvm-cov	Two tools for generating test coverage reports in Go and Rust	Appendix D.4 and D.6
cargo-audit	A tool for checking for vulnerable dependencies	Appendix D.5
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix D.7

Areas of Focus

Our automated testing and verification focused on the following:

- Identification of general code quality issues and unidiomatic code patterns
- Identification of untested code regions with coverage reports

Test Results

The results of this focused testing are detailed below.

Rust Library

Property	Tool	Result
`cast-possible-truncation` and `cast-possible-wrap`	Clippy	TOB-ZKTRIE-19

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	We found one high-severity issue related to integer casting, TOB-ZKTRIE-19, that allows an attacker to create nodes from different byte arrays that hash to the same value. The same issue also allows an attacker to crash the program.	**Moderate**
Complexity Management	The code is fairly well structured and separated into files with related functions grouped together. However, the control flow in some cases is complex and spread across several files. For example, the Rust `ZkTrie::prove` method calls `TrieProve` through `cgo`, which calls `ZkTrie.Prove`, which then calls a callback defined in `TrieProve`, which calls a C function that calls a Rust callback. We found that the codebase has redundant representations of nodes and different incompatible ways to generate proofs (TOB-ZKTRIE-3). The data model differs between the Rust and Go APIs: in Rust, the data model closely matches the specification document, while in Go, the zkTrie has a more generic key-value store API. This API mismatch led to a type confusion vulnerability (TOB-ZKTRIE-18).	**Weak**
Cryptography and Key Management	We found two high-severity issues related to the misuse of hash functions in the context of Merkle trees (TOB-ZKTRIE-1, TOB-ZKTRIE-5) that allows an attacker to create proof forgeries.	**Weak**
Data Handling	We identified many findings related to unexpected or malicious inputs (TOB-ZKTRIE-2, TOB-ZKTRIE-10,	**Weak**

	TOB-ZKTRIE-11, TOB-ZKTRIE-12, and TOB-ZKTRIE-19) that allow an attacker to crash the program. We also found that other externally facing functions do not validate their inputs (TOB-ZKTRIE-14).	
Documentation	The codebase includes a specification document that describes some data schema and a prose description of the insertion and deletion algorithms. However, several parts of the code, including the Rust API and the database API, are not documented. All functions should be documented, along with the roles of their arguments and potential function requirements. This is of particular importance for the public API functions in both the Go and Rust libraries.	**Weak**
Memory Safety and Error Handling	We found memory safety issues in the `cgo` interface related to concurrency (TOB-ZKTRIE-12, TOB-ZKTRIE-15, TOB-ZKTRIE-16, and TOB-ZKTRIE-18). We also found inconsistent error handling and propagation (TOB-ZKTRIE-2, TOB-ZKTRIE-17, and appendix C).	**Weak**
Testing and Verification	We found that the codebase includes unit tests with fairly good coverage. However, we did find some issues that would have been found with more extensive unit testing (e.g., TOB-ZKTRIE-4). We recommend using the test coverage tools described in appendix D to achieve full coverage of all code branches. Additionally, negative testing and fuzz testing could be used to find most findings that result in a program panic. Appendix E and appendix F present two ways of testing the codebase. All publicly facing functionality should be extensively tested using fuzz testing. Other than these tests, static analysis tools like `golangci-lint` and Semgrep should be integrated into Scroll's CI/CD pipeline.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lack of domain separation allows proof forgery	Cryptography	High
2	Lack of proof validation causes denial of service on the verifier	Data Validation	Medium
3	Two incompatible ways to generate proofs	Data Validation	Informational
4	BuildZkTrieProof does not populate NodeAux.Value	Testing	Low
5	Leaf nodes with different values may have the same hash	Cryptography	High
6	Empty UpdatePreimage function body	Auditing and Logging	Informational
7	CanonicalValue is not canonical	Cryptography	Informational
8	ToSecureKey and ToSecureKeyBytes implicitly truncate the key	Data Validation	Informational
9	Unused key argument on the bridge_prove_write function	Auditing and Logging	Informational
10	The PreHandlingElems function panics with an empty elems array	Data Validation	Medium
11	The hash_external function panics with integers larger than 32 bytes	Data Validation	Low
12	Mishandling of cgo.Handles causes runtime errors	Data Validation	Medium

13	Unnecessary unsafe pointer manipulation in Node.Data()	Undefined Behavior	**Informational**
14	NewNodeFromBytes does not fully validate its input	Data Validation	**Informational**
15	init_hash_scheme is not thread-safe	Undefined Behavior	**Informational**
16	Safe-Rust ZkMemoryDb interface is not thread-safe	Undefined Behavior	**High**
17	Some Node functions return the zero hash instead of errors	Error Reporting	**Informational**
18	get_account can read past the buffer	Data Exposure	**High**
19	Unchecked usize to c_int casts allow hash collisions by length misinterpretation	Data Validation	**High**

Detailed Findings

1. Lack of domain separation allows proof forgery	
Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-ZKTRIE-1
Target: `trie/zk_trie_node.go`	

Description

Merkle trees are nested tree data structures in which the hash of each branch node depends upon the hashes of its children. The hash of each node is then assumed to uniquely represent the subtree of which that node is a root. However, that assumption may be false if a leaf node can have the same hash as a branch node. A general method for preventing leaf and branch nodes from colliding in this way is *domain separation*. That is, given a hash function H, define the hash of a leaf to be $H(f(leaf\_data))$ and the hash of a branch to be $H(g(branch\_data))$, where f and g are encoding functions that can never return the same result (perhaps because f's return values all start with the byte 0 and g's all start with the byte 1). Without domain separation, a malicious entity may be able to insert a leaf into the tree that can be later used as a branch in a Merkle path.

In `zktrie`, the hash for a node is defined by the `NodeHash` method, shown in figure 1.1. As shown in the highlighted portions, the hash of a branch node is `HashElems(n.ChildL,n.ChildR)`, while the hash of a leaf node is `HashElems(1,n.NodeKey,n.valueHash)`.

```
// LeafHash computes the key of a leaf node given the hIndex and hValue of the
// entry of the leaf.
func LeafHash(k, v *zkt.Hash) (*zkt.Hash, error) {
	return zkt.HashElems(big.NewInt(1), k.BigInt(), v.BigInt())
}

// NodeHash computes the hash digest of the node by hashing the content in a
// specific way for each type of node.  This key is used as the hash of the
// Merkle tree for each node.
func (n *Node) NodeHash() (*zkt.Hash, error) {
	if n.nodeHash == nil { // Cache the key to avoid repeated hash computations.
		// NOTE: We are not using the type to calculate the hash!
		switch n.Type {
		case NodeTypeParent: // H(ChildL || ChildR)
			var err error
			n.nodeHash, err = zkt.HashElems(n.ChildL.BigInt(), n.ChildR.BigInt())
```

```
                        if err != nil {
                                return nil, err
                        }
                case NodeTypeLeaf:
                        var err error
                        n.valueHash, err = zkt.PreHandlingElems(n.CompressedFlags,
n.ValuePreimage)
                        if err != nil {
                                return nil, err
                        }
                        n.nodeHash, err = LeafHash(n.NodeKey, n.valueHash)
                        if err != nil {
                                return nil, err
                        }

                case NodeTypeEmpty: // Zero
                        n.nodeHash = &zkt.HashZero
                default:
                        n.nodeHash = &zkt.HashZero
                }
        }
        return n.nodeHash, nil
}
```

Figure 1.1: NodeHash and LeafHash (zktrie/trie/zk_trie_node.go#118–156)

The function `HashElems` used here performs recursive hashing in a binary-tree fashion. For the purpose of this finding, the key property is that `HashElems(1,k,v) == H(H(1,k),v)` and `HashElems(n.ChildL,n.ChildR) == H(n.ChildL,n.ChildR)`, where H is the global two-input, one-output hash function. Therefore, a branch node `b` and a leaf node `l` where `b.ChildL == H(1,l.NodeKey)` and `b.ChildR == l.valueHash` will have equal hash values.

This allows proof forgery and, for example, a malicious entity to insert a key that can be proved to be both present and nonexistent in the tree, as illustrated by the proof-of-concept test in figure 1.2.

```
func TestMerkleTree_ForgeProof(t *testing.T) {
        zkTrie := newTestingMerkle(t, 10)

        t.Run("Testing for malicious proofs", func(t *testing.T) {

                // Find two distinct values k1,k2 such that the first step of
                // the path has the sibling on the LEFT (i.e., path[0] ==
                // false)
                k1, k2 := (func() (zkt.Byte32, zkt.Byte32) {
                        k1 := zkt.Byte32{1}
                        k2 := zkt.Byte32{2}
                        k1_hash, _ := k1.Hash()
                        k2_hash, _ := k2.Hash()
```

```
			for !getPath(1, zkt.NewHashFromBigInt(k1_hash)[:])[0] {
				for i := len(k1); i > 0; i -= 1 {
					k1[i-1] += 1
					if k1[i-1] != 0 {
						break
					}
				}
				k1_hash, _ = k1.Hash()
			}

			for k1 == k2 || !getPath(1, 
zkt.NewHashFromBigInt(k2_hash)[:])[0] {
				for i := len(k2); i > 0; i -= 1 {
					k2[i-1] += 1
					if k2[i-1] != 0 {
						break
					}
				}
				k2_hash, _ = k2.Hash()
			}

			return k1, k2
		})()

		k1_hash_int, _ := k1.Hash()
		k2_hash_int, _ := k2.Hash()
		k1_hash := zkt.NewHashFromBigInt(k1_hash_int)
		k2_hash := zkt.NewHashFromBigInt(k2_hash_int)

		// create a dummy value for k2, and use that to craft a
		// malicious value for k1
		k2_value := (&[2]zkt.Byte32{{2}})[:]
		k1_value, _ := NewLeafNode(k2_hash, 1, k2_value).NodeHash()

		k1_value_array := 
[]zkt.Byte32{*zkt.NewByte32FromBytes(k1_value.Bytes())}

		// insert k1 into the trie with the malicious value
		assert.Nil(t, zkTrie.TryUpdate(zkt.NewHashFromBigInt(k1_hash_int), 0,
			k1_value_array))

		getNode := func(hash *zkt.Hash) (*Node, error) {
			return zkTrie.GetNode(hash)
		}

		// query an inclusion proof for k1
		k1Proof, _, err := BuildZkTrieProof(zkTrie.rootHash, k1_hash_int, 10, 
getNode)
		assert.Nil(t, err)
		assert.True(t, k1Proof.Existence)

		// check that inclusion proof against our root hash
```

```
            k1_val_hash, _ := NewLeafNode(k1_hash, 0, k1_value_array).NodeHash()
            k1Proof_root, _ := k1Proof.Verify(k1_val_hash, k1_hash)
            assert.Equal(t, k1Proof_root, zkTrie.rootHash)

            // forge a non-existence proof
            fakeNonExistProof := *k1Proof
            fakeNonExistProof.Existence = false

            // The new non-existence proof needs one extra level, where
            // the sibling hash is H(1,k1_hash)
            fakeNonExistProof.depth += 1
            zkt.SetBitBigEndian(fakeNonExistProof.notempties[:],
fakeNonExistProof.depth-1)
            fakeSibHash, _ := zkt.HashElems(big.NewInt(1), k1_hash_int)
            fakeNonExistProof.Siblings = append(fakeNonExistProof.Siblings,
fakeSibHash)

            // Construct the NodeAux details for the malicious leaf
            k2_value_hash, _ := zkt.PreHandlingElems(1, k2_value)
            k2_nodekey := zkt.NewHashFromBigInt(k2_hash_int)
            fakeNonExistProof.NodeAux = &NodeAux{Key: k2_nodekey, Value:
k2_value_hash}

            // Check our non-existence proof against the root hash
            fakeNonExistProof_root, _ := fakeNonExistProof.Verify(k1_val_hash,
k1_hash)
            assert.Equal(t, fakeNonExistProof_root, zkTrie.rootHash)

            // fakeNonExistProof and k1Proof prove opposite things. k1
            // is both in and not-in the tree!
            assert.NotEqual(t, fakeNonExistProof.Existence, k1Proof.Existence)
      })
}
```

Figure 1.2: A proof-of-concept test case for proof forgery

Exploit Scenario

Suppose Alice uses the `zktrie` to implement the Ethereum account table in a zkEVM-based bridge with trustless state updates. Bob submits a transaction that inserts specially crafted account data into some position in that tree. At a later time, Bob submits a transaction that depends on the result of an account table lookup. Bob generates two contradictory Merkle proofs and uses those proofs to create two zkEVM execution proofs that step to different final states. By submitting one proof each to the opposite sides of the bridge, Bob causes state divergence and a loss of funds.

Recommendations

Short term, modify `NodeHash` to domain-separate leaves and branches, such as by changing the branch hash to `zkt.HashElems(big.NewInt(2),n.ChildL.BigInt(), n.ChildR.BigInt())`.

Long term, fully document all data structure designs and requirements, and review all assumptions to ensure that they are well founded.

2. Lack of proof validation causes denial of service on the verifier

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ZKTRIE-2
Target: `trie/zk_trie_impl.go`	

Description

The Merkle tree proof verifier assumes several well-formedness properties about the received proof and node arguments. If at least one of these properties is violated, the verifier will have a runtime error.

The first property that must hold is that the node associated with the Merkle proof must be a leaf node (i.e., must contain a non-nil `NodeKey` field). If this is not the case, computing the `rootFromProof` for a `nil` `NodeKey` will cause a panic when computing the `getPath` function.

Secondly, the `Proof` fields must be guaranteed to be consistent with the other fields. Assuming that the proof depth is correct will cause out-of-bounds accesses to both the `NodeKey` and the `notempties` field. Finally, the `Siblings` array length should also be validated; for example, the `VerifyProofZkTrie` will panic due to an out-of-bounds access if the `proof.Siblings` field is empty (highlighted in yellow in the `rootFromProof` function).

```go
// VerifyProof verifies the Merkle Proof for the entry and root.
func VerifyProofZkTrie(rootHash *zkt.Hash, proof *Proof, node *Node) bool {
	nodeHash, err := node.NodeHash()
	if err != nil {
		return false
	}

	rootFromProof, err := proof.Verify(nodeHash, node.NodeKey)
	if err != nil {
		return false
	}
	return bytes.Equal(rootHash[:], rootFromProof[:])
}

// Verify the proof and calculate the root, nodeHash can be nil when try to verify
// a nonexistent proof
func (proof *Proof) Verify(nodeHash, nodeKey *zkt.Hash) (*zkt.Hash, error) {
	if proof.Existence {
		if nodeHash == nil {
				return nil, ErrKeyNotFound
			}
			return proof.rootFromProof(nodeHash, nodeKey)
		} else {
			if proof.NodeAux == nil {
				return proof.rootFromProof(&zkt.HashZero, nodeKey)
			} else {
				if bytes.Equal(nodeKey[:], proof.NodeAux.Key[:]) {
					return nil, fmt.Errorf("non-existence proof being checked 
against hIndex equal to nodeAux")
				}
				midHash, err := LeafHash(proof.NodeAux.Key, proof.NodeAux.Value)
				if err != nil {
					return nil, err
				}
				return proof.rootFromProof(midHash, nodeKey)
			}
		}

}

func (proof *Proof) rootFromProof(nodeHash, nodeKey *zkt.Hash) (*zkt.Hash, error) {
	var err error

	sibIdx := len(proof.Siblings) - 1
	path := getPath(int(proof.depth), nodeKey[:])
	var siblingHash *zkt.Hash
	for lvl := int(proof.depth) - 1; lvl >= 0; lvl-- {
		if zkt.TestBitBigEndian(proof.notempties[:], uint(lvl)) {
			siblingHash = proof.Siblings[sibIdx]
			sibIdx--
		} else {
			siblingHash = &zkt.HashZero
		}
		if path[lvl] {
			nodeHash, err = NewParentNode(siblingHash, nodeHash).NodeHash()
			if err != nil {
				return nil, err
			}
		} else {
			nodeHash, err = NewParentNode(nodeHash, siblingHash).NodeHash()
			if err != nil {
				return nil, err
			}
		}
	}
	return nodeHash, nil
}
```

Figure 2.1: `zktrie/trie/zk_trie_impl.go#595–660`

Exploit Scenario
An attacker crafts an invalid proof that causes the proof verifier to crash, causing a denial of service in the system.

Recommendations
Short term, validate the proof structure before attempting to use its values. Add fuzz testing to the `VerifyProofZkTrie` function.

Long term, add extensive tests and fuzz testing to functions interfacing with attacker-controlled values.

3. Two incompatible ways to generate proofs

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ZKTRIE-3
Target: `trie/{zk_trie.go, zk_trie_impl.go}`	

Description

There are two incompatible ways to generate proofs.

The first implementation (figure 3.1) writes to a given callback, effectively returning `[]bytes`. It does not have a companion verification function; it has only positive tests (`zktrie/trie/zk_trie_test.go#L93-L125`); and it is accessible from the C function `TrieProve` and the Rust function `prove`.

The second implementation (figure 3.2) returns a pointer to a `Proof` struct. It has a companion verification function (`zktrie/trie/zk_trie_impl.go#L595-L632`); it has positive and negative tests (`zktrie/trie/zk_trie_impl_test.go#L484-L537`); and it is not accessible from C or Rust.

```
// Prove is a simlified calling of ProveWithDeletion
func (t *ZkTrie) Prove(key []byte, fromLevel uint, writeNode func(*Node) error)
error {
       return t.ProveWithDeletion(key, fromLevel, writeNode, nil)
}

// ProveWithDeletion constructs a merkle proof for key. The result contains all
encoded nodes
// on the path to the value at key. The value itself is also included in the last
// node and can be retrieved by verifying the proof.
//
// If the trie does not contain a value for key, the returned proof contains all
// nodes of the longest existing prefix of the key (at least the root node), ending
// with the node that proves the absence of the key.
//
// If the trie contain value for key, the onHit is called BEFORE writeNode being
called,
// both the hitted leaf node and its sibling node is provided as arguments so caller
// would receive enough information for launch a deletion and calculate the new root
// base on the proof data
// Also notice the sibling can be nil if the trie has only one leaf
func (t *ZkTrie) ProveWithDeletion(key []byte, fromLevel uint, writeNode func(*Node)
error, onHit func(*Node, *Node)) error {
       [...]
}
```

Figure 3.1: The first way to generate proofs (`zktrie/trie/zk_trie.go#143–164`)

```go
// Proof defines the required elements for a MT proof of existence or
// non-existence.
type Proof struct {
	// existence indicates wether this is a proof of existence or
	// non-existence.
	Existence bool
	// depth indicates how deep in the tree the proof goes.
	depth uint
	// notempties is a bitmap of non-empty Siblings found in Siblings.
	notempties [zkt.HashByteLen - proofFlagsLen]byte
	// Siblings is a list of non-empty sibling node hashes.
	Siblings []*zkt.Hash
	// NodeAux contains the auxiliary information of the lowest common ancestor
	// node in a non-existence proof.
	NodeAux *NodeAux
}

// BuildZkTrieProof prove uniformed way to turn some data collections into Proof
struct
func BuildZkTrieProof(rootHash *zkt.Hash, k *big.Int, lvl int, getNode func(key
*zkt.Hash) (*Node, error)) (*Proof,
	*Node, error) {
	[...]
}
```

Figure 3.2: The second way to generate proofs (`zktrie/trie/zk_trie_impl.go#531–551`)

Recommendations

Short term, decide on one implementation and remove the other implementation.

Long term, ensure full test coverage in the chosen implementation; ensure the implementation has both positive and negative testing; and add fuzz testing to the proof verification routine.

4. BuildZkTrieProof does not populate NodeAux.Value	
Severity: **Low**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-ZKTRIE-4
Target: `trie/zk_trie_impl.go`	

Description

A nonexistence proof for some key `k` in a Merkle tree is a Merkle path from the root of the tree to a subtree, which would contain `k` if it were present but which instead is either an empty subtree or a subtree with a single leaf `k2` where `k != k2`. In the `zktrie` codebase, that second case is handled by the `NodeAux` field in the `Proof` struct, as illustrated in figure 4.1.

```
// Verify the proof and calculate the root, nodeHash can be nil when try to verify
// a nonexistent proof
func (proof *Proof) Verify(nodeHash, nodeKey *zkt.Hash) (*zkt.Hash, error) {
	if proof.Existence {
		if nodeHash == nil {
			return nil, ErrKeyNotFound
		}
		return proof.rootFromProof(nodeHash, nodeKey)
	} else {
		if proof.NodeAux == nil {
			return proof.rootFromProof(&zkt.HashZero, nodeKey)
		} else {
			if bytes.Equal(nodeKey[:], proof.NodeAux.Key[:]) {
				return nil, fmt.Errorf("non-existence proof being checked
against hIndex equal to nodeAux")
			}
			midHash, err := LeafHash(proof.NodeAux.Key, proof.NodeAux.Value)
			if err != nil {
				return nil, err
			}
			return proof.rootFromProof(midHash, nodeKey)
		}
	}

}
```

Figure 4.1: The `Proof.Verify` method (`zktrie/trie/zk_trie_impl.go#609–632`)

When a non-inclusion proof is generated, the `BuildZkTrieProof` function looks up the other leaf node and uses its `NodeKey` and `valueHash` fields to populate the `Key` and `Value` fields of `NodeAux`, as shown in figure 4.2. However, the `valueHash` field of this node may be `nil`, causing `NodeAux.Value` to be `nil` and causing proof verification to crash with a nil pointer dereference error, which can be triggered by the test case shown in figure 4.3.

```
n, err := getNode(nextHash)
if err != nil {
	return nil, nil, err
}
switch n.Type {
case NodeTypeEmpty:
	return p, n, nil
case NodeTypeLeaf:
	if bytes.Equal(kHash[:], n.NodeKey[:]) {
		p.Existence = true
		return p, n, nil
	}
	// We found a leaf whose entry didn't match hIndex
	p.NodeAux = &NodeAux{Key: n.NodeKey, Value: n.valueHash}
	return p, n, nil
```

Figure 4.2: Populating NodeAux (`zktrie/trie/zk_trie_impl.go#560–574`)

```
func TestMerkleTree_GetNonIncProof(t *testing.T) {
	zkTrie := newTestingMerkle(t, 10)

	t.Run("Testing for non-inclusion proofs", func(t *testing.T) {
		k := zkt.Byte32{1}
		k_value := (&[1]zkt.Byte32{{1}})[:]

		k_other := zkt.Byte32{2}

		k_hash_int, _ := k.Hash()
		k_other_hash_int, _ := k_other.Hash()

		k_hash := zkt.NewHashFromBigInt(k_hash_int)
		k_other_hash := zkt.NewHashFromBigInt(k_other_hash_int)

		assert.Nil(t, zkTrie.TryUpdate(k_hash, 0, k_value))

		getNode := func(hash *zkt.Hash) (*Node, error) {
			return zkTrie.GetNode(hash)
		}
		proof, _, err := BuildZkTrieProof(zkTrie.rootHash, k_other_hash_int,
10, getNode)
		assert.Nil(t, err)
		assert.False(t, proof.Existence)

		proof_root, _ := proof.Verify(nil, k_other_hash)
		assert.Equal(t, proof_root, zkTrie.rootHash)
	})
}
```

Figure 4.3: A test case that will crash with a nil dereference of `NodeAux.Value`

Adding a call to `n.NodeHash()` inside `BuildZkTrieProof`, as shown in figure 4.4, fixes this problem.

```
n, err := getNode(nextHash)
if err != nil {
	return nil, nil, err
}


switch n.Type {
case NodeTypeEmpty:
	return p, n, nil
case NodeTypeLeaf:
	if bytes.Equal(kHash[:], n.NodeKey[:]) {
		p.Existence = true
		return p, n, nil
	}
	// We found a leaf whose entry didn't match hIndex
	p.NodeAux = &NodeAux{Key: n.NodeKey, Value: n.valueHash}
	return p, n, nil
```

Figure 4.4: Adding the highlighted `n.NodeHash()` call fixes this problem.
(`zktrie/trie/zk_trie_impl.go#560–574`)

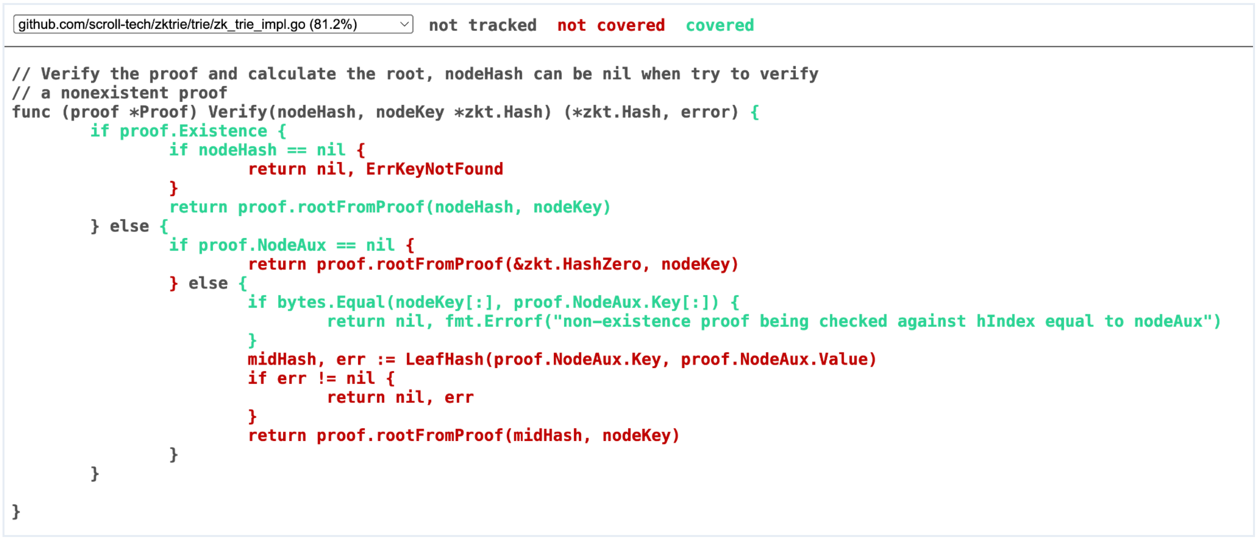
Exploit Scenario

An adversary or ordinary user requests that the software generate and verify a non-inclusion proof, and the software crashes, leading to the loss of service.

Recommendations

Short term, fix `BuildZkTrieProof` by adding a call to `n.NodeHash()`, as described above.

Long term, ensure that all major code paths in important functions, such as proof generation and verification, are tested. The Go coverage analysis report generated by the command `go test -cover -coverprofile c.out && go tool cover -html=c.out` shows that this branch in `Proof.Verify` is not currently tested:


github.com/scroll-tech/zktrie/trie/zk_trie_impl.go (81.2%) not tracked not covered covered

```
// Verify the proof and calculate the root, nodeHash can be nil when try to verify
// a nonexistent proof
func (proof *Proof) Verify(nodeHash, nodeKey *zkt.Hash) (*zkt.Hash, error) {
	if proof.Existence {
		if nodeHash == nil {
			return nil, ErrKeyNotFound
		}
		return proof.rootFromProof(nodeHash, nodeKey)
	} else {
		if proof.NodeAux == nil {
			return proof.rootFromProof(&zkt.HashZero, nodeKey)
		} else {
			if bytes.Equal(nodeKey[:], proof.NodeAux.Key[:]) {
				return nil, fmt.Errorf("non-existence proof being checked against hIndex equal to nodeAux")
			}
			midHash, err := LeafHash(proof.NodeAux.Key, proof.NodeAux.Value)
			if err != nil {
				return nil, err
			}
			return proof.rootFromProof(midHash, nodeKey)
		}
	}

}
```

Figure 4.5: The Go coverage analysis report

5. Leaf nodes with different values may have the same hash

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-ZKTRIE-5
Target: `trie/zk_trie_node.go`, `types/util.go`	

Description

The hash value of a leaf node is derived from the hash of its key and its value. A leaf node's value comprises up to 256 32-byte fields, and that value's hash is computed by passing those fields to the `HashElems` function. `HashElems` hashes these fields in a Merkle tree–style binary tree pattern, as shown in figure 5.1.

```go
func HashElems(fst, snd *big.Int, elems ...*big.Int) (*Hash, error) {
	l := len(elems)
	baseH, err := hashScheme([]*big.Int{fst, snd})
	if err != nil {
		return nil, err
	}
	if l == 0 {
		return NewHashFromBigInt(baseH), nil
	} else if l == 1 {
		return HashElems(baseH, elems[0])
	}
	tmp := make([]*big.Int, (l+1)/2)
	for i := range tmp {
		if (i+1)*2 > l {
			tmp[i] = elems[i*2]
		} else {
			h, err := hashScheme(elems[i*2 : (i+1)*2])
			if err != nil {
				return nil, err
			}
			tmp[i] = h
		}
	}

	return HashElems(baseH, tmp[0], tmp[1:]...)
}
```

Figure 5.1: Binary-tree hashing in HashElems (`zktrie/types/util.go#9–36`)

However, `HashElems` does not include the number of elements being hashed, so leaf nodes with different values may have the same hash, as illustrated in the proof-of-concept test case shown in figure 5.2.

```
func TestMerkleTree_MultiValue(t *testing.T) {
	t.Run("Testing for value collisions", func(t *testing.T) {
		k := zkt.Byte32{1}
		k_hash_int, _ := k.Hash()
		k_hash := zkt.NewHashFromBigInt(k_hash_int)

		value1 := (&[3]zkt.Byte32{{1}, {2}, {3}})[:]
		value1_hash, _ := NewLeafNode(k_hash, 0, value1).NodeHash()
		first2_hash, _ := zkt.PreHandlingElems(0, value1[:2])
		value2 := (&[2]zkt.Byte32{*zkt.NewByte32FromBytes(first2_hash.Bytes()),
{3}})[:]
		value2_hash, _ := NewLeafNode(k_hash, 0, value2).NodeHash()

		assert.NotEqual(t, value1, value2)
		assert.NotEqual(t, value1_hash, value2_hash)
	})
}
```

Figure 5.2: A proof-of-concept test case for value collisions

Exploit Scenario

An adversary inserts a maliciously crafted value into the tree and then creates a proof for a different, colliding value. This violates the security requirements of a Merkle tree and may lead to incorrect behavior such as state divergence.

Recommendations

Short term, modify `PrehandlingElems` to prefix the `ValuePreimage` array with its length before being passed to `HashElems`.

Long term, document and review all uses of hash functions to ensure that they commit to their inputs.

6. Empty UpdatePreimage function body

Severity: **Informational**	Difficulty: **N/A**
Type: Auditing and Logging	Finding ID: TOB-ZKTRIE-6
Target: `trie/zk_trie_database.go`	

Description

The `UpdatePreimage` function implementation for the `Database` receiver type is empty. Instead of an empty function body, the function should either panic with an `unimplemented` message or a message that is logged. This would prevent the function from being used without any warning.

```
func (db *Database) UpdatePreimage([]byte, *big.Int) {}
```

Figure 6.1: `zktrie/trie/zk_trie_database.go#19`

Recommendations

Short term, add an `unimplemented` message to the function body, through either a panic or message logging.

7. CanonicalValue is not canonical

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-ZKTRIE-7
Target: `trie/zk_trie_node.go`	

Description

The `CanonicalValue` function does not uniquely generate a representation of `Node` structures, allowing different `Nodes` with the same `CanonicalValue`, and two nodes with the same `NodeHash` but different `CanonicalValues`.

`ValuePreimages` in a `Node` can be either uncompressed or compressed (by hashing); the `CompressedFlags` value indicates which data is compressed.

Only the first 24 fields can be compressed, so `CanonicalValue` truncates `CompressedFlags` to the first 24 bits. But `NewLeafNode` accepts any `uint32` for the `CompressedFlags` field of a `Node`. Figure 7.3 shows how this can be used to construct two different `Node` structs that have the same `CanonicalValue`.

```
// CanonicalValue returns the byte form of a node required to be persisted, and
strip unnecessary fields
// from the encoding (current only KeyPreimage for Leaf node) to keep a minimum size
for content being
// stored in backend storage
func (n *Node) CanonicalValue() []byte {
	switch n.Type {
	case NodeTypeParent: // {Type || ChildL || ChildR}
		bytes := []byte{byte(n.Type)}
		bytes = append(bytes, n.ChildL.Bytes()...)
		bytes = append(bytes, n.ChildR.Bytes()...)
		return bytes
	case NodeTypeLeaf: // {Type || Data...}
		bytes := []byte{byte(n.Type)}
		bytes = append(bytes, n.NodeKey.Bytes()...)
		tmp := make([]byte, 4)
		compressedFlag := (n.CompressedFlags << 8) +
uint32(len(n.ValuePreimage))
		binary.LittleEndian.PutUint32(tmp, compressedFlag)
		bytes = append(bytes, tmp...)
		for _, elm := range n.ValuePreimage {
			bytes = append(bytes, elm[:]...)
		}
		bytes = append(bytes, 0)
		return bytes
```

```
        case NodeTypeEmpty: // { Type }
                return []byte{byte(n.Type)}
        default:
                return []byte{}
        }
}
```

Figure 7.1: This figure shows the CanonicalValue computation. The highlighted code assumes that CompressedFlags is 24 bits. (zktrie/trie/zk_trie_node.go#187–214)

```
// NewLeafNode creates a new leaf node.
func NewLeafNode(k *zkt.Hash, valueFlags uint32, valuePreimage []zkt.Byte32) *Node {
        return &Node{Type: NodeTypeLeaf, NodeKey: k, CompressedFlags: valueFlags,
ValuePreimage: valuePreimage}
}
```

Figure 7.2: Node construction in NewLeafNode (zktrie/trie/zk_trie_node.go#55–58)

```
// CanonicalValue implicitly truncates CompressedFlags to 24 bits. This test should
ideally fail.
func TestZkTrie_CanonicalValue1(t *testing.T) {
        key, err :=
hex.DecodeString("0000000000000000000000000000000000000000000000000000000000000000")
        assert.NoError(t, err)
        vPreimage := []zkt.Byte32{{0}}

        k := zkt.NewHashFromBytes(key)

        vFlag0 := uint32(0x00ffffff)
        vFlag1 := uint32(0xffffffff)

        lf0 := NewLeafNode(k, vFlag0, vPreimage)
        lf1 := NewLeafNode(k, vFlag1, vPreimage)

        // These two assertions should never simultaneously pass.
        assert.True(t, lf0.CompressedFlags != lf1.CompressedFlags)
        assert.True(t, reflect.DeepEqual(lf0.CanonicalValue(), lf1.CanonicalValue()))
}
```

Figure 7.3: A test showing that one can construct different nodes with the same CanonicalValue

```
// PreHandlingElems turn persisted byte32 elements into field arrays for our
hashElem
// it also has the compressed byte32
func PreHandlingElems(flagArray uint32, elems []Byte32) (*Hash, error) {

        ret := make([]*big.Int, len(elems))
        var err error

        for i, elem := range elems {
                if flagArray&(1<<i) != 0 {
                        ret[i], err = elem.Hash()
                        if err != nil {
                                return nil, err
                        }
                } else {
                        ret[i] = new(big.Int).SetBytes(elem[:])
                }
        }

        if len(ret) < 2 {
                return NewHashFromBigInt(ret[0]), nil
        }

        return HashElems(ret[0], ret[1], ret[2:]...)

}
```

Figure 7.4: The subroutine called in NodeHash that hashes uncompressed elements (`zktrie/types/util.go#38–62`)

Furthermore, `CanonicalValue` and `NodeHash` are inconsistent in their processing of uncompressed values. `CanonicalValue` uses them directly, while `NodeHash` hashes them. Figure 7.5 shows how this can be used to construct two `Node` structs that have the same `NodeHash` but different `CanonicalValues`.

```
// CanonicalValue and NodeHash are not consistent
func TestZkTrie_CanonicalValue2(t *testing.T) {
        t.Run("Testing for value collisions", func(t *testing.T) {
                k := zkt.Byte32{1}
                k_hash_int, _ := k.Hash()
                k_hash := zkt.NewHashFromBigInt(k_hash_int)

                value1 := (&[2]zkt.Byte32{*zkt.NewByte32FromBytes(k_hash.Bytes()),
{3}})[:]
                value2 := (&[2]zkt.Byte32{{1}, {3}})[:]

                leaf1 := NewLeafNode(k_hash, 0, value1)
                leaf2 := NewLeafNode(k_hash, 1, value2)

                leaf1_node_hash, _ := leaf1.NodeHash()
                leaf2_node_hash, _ := leaf2.NodeHash()
                assert.Equal(t, leaf1_node_hash, leaf2_node_hash)

                leaf1_canonical := leaf1.CanonicalValue()
                leaf2_canonical := leaf2.CanonicalValue()
                assert.NotEqual(t, leaf1_canonical, leaf2_canonical)
        })
}
```

Figure 7.5: A test showing that `CanonicalValue` and `NodeHash` are inconsistent

Recommendations
Short term, have `CanonicalValue` validate all assumptions, and make `CanonicalValue` and `NodeHash` consistent.

Long term, document all assumptions and use Go's type system to enforce them.

8. ToSecureKey and ToSecureKeyBytes implicitly truncate the key

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ZKTRIE-8
Target: `types/util.go`	

Description

`ToSecureKey` and `ToSecureKeyBytes` accept a key of arbitrary length but implicitly truncate it to 32 bytes.

`ToSecureKey` makes an underlying call to `NewByte32FromBytesPaddingZero` that truncates the key to its first 32 bytes. The `ToSecureKeyBytes` function also truncates the key because it calls `ToSecureKey`.

```
// ToSecureKey turn the byte key into the integer represent of "secured" key
func ToSecureKey(key []byte) (*big.Int, error) {
	word := NewByte32FromBytesPaddingZero(key)
	return word.Hash()
}
```

Figure 8.1: `ToSecureKey` accepts a key of arbitrary length. (`zktrie/types/util.go#93–97`)

```
// create bytes32 with zeropadding to shorter bytes, or truncate it
func NewByte32FromBytesPaddingZero(b []byte) *Byte32 {
	byte32 := new(Byte32)
	copy(byte32[:], b)
	return byte32
}
```

Figure 8.2: But `NewByte32FromBytesPaddingZero` truncates the key to the first 32 bytes. (`zktrie/types/byte32.go#35–40`)

```
// ToSecureKeyBytes turn the byte key into a 32-byte "secured" key, which
represented a big-endian integer
func ToSecureKeyBytes(key []byte) (*Byte32, error) {
	k, err := ToSecureKey(key)
	if err != nil {
		return nil, err
	}

	return NewByte32FromBytes(k.Bytes()), nil
}
```

Figure 8.3: `ToSecureKeyBytes` accepts a key of arbitrary length and calls `ToSecureKey` on it. (`zktrie/types/util.go#99–107`)

```
// zkt.ToSecureKey implicitly truncates keys to 32 bytes. This test should ideally
fail.
func TestZkTrie_ToSecureKeyTruncation(t *testing.T) {
	key1, err :=
hex.DecodeString("0000000000000000000000000000000000000000000000000000000000000011
")
	assert.NoError(t, err)
	key2, err :=
hex.DecodeString("0000000000000000000000000000000000000000000000000000000000000022
")
	assert.NoError(t, err)
	assert.NotEqual(t, key1, key2)

	skey1, err := zkt.ToSecureKey(key1)
	assert.NoError(t, err) // This should fail
	skey2, err := zkt.ToSecureKey(key2)
	assert.NoError(t, err) // This should fail
	assert.True(t, skey1.Cmp(skey2) == 0) // If above don't fail, this should
fail
}
```

Figure 8.4: A test showing the truncation of keys longer than 32 bytes

Recommendations

Short term, fix the `ToSecureKey` and `ToSecureKeyBytes` functions so that they do not truncate keys that are longer than 32 bytes, and instead hash all the bytes. If this behavior is not desired, ensure that the functions return an error if given a key longer than 32 bytes.

Long term, add fuzz tests to public interfaces like `TryGet`, `TryUpdate`, and `TryDelete`.

9. Unused key argument on the bridge_prove_write function

Severity: **Informational**	Difficulty: **N/A**
Type: Auditing and Logging	Finding ID: TOB-ZKTRIE-9
Target: `c.go`	

Description

The `bridge_prove_write` function implementation does not use the `key` argument.

```
void bridge_prove_write(proveWriteF f, unsigned char* key, unsigned char* val, int
size, void* param){
	f(val, size, param);
}
```

Figure 9.1: `zktrie/c.go#17–19`

This function is always called with a `nil` value:

```
	err = tr.Prove(s_key.Bytes(), 0, func(n *trie.Node) error {

		dt := n.Value()

		C.bridge_prove_write(
			C.proveWriteF(callback),
			nil, //do not need to prove node key
			(*C.uchar)(&dt[0]),
			C.int(len(dt)),
			cb_param,
		)

		return nil
	})
	if err != nil {
		return C.CString(err.Error())
	}

	tailingLine := trie.ProofMagicBytes()
	C.bridge_prove_write(
		C.proveWriteF(callback),
		nil, //do not need to prove node key
		(*C.uchar)(&tailingLine[0]),
		C.int(len(tailingLine)),
		cb_param,
	)
```

```
        return nil
}
```

Figure 9.2: *`zktrie/lib.go#263–292`*

Recommendations

Short term, document the intended behavior and the role and requirements of each function. Decide whether to remove the unused argument or document why it is currently unused in the implementation.

10. The PreHandlingElems function panics with an empty elems array

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ZKTRIE-10
Target: `types/util.go`, `trie/zk_trie_node.go`	

Description

The `PreHandlingElems` function experiences a runtime error when the `elems` array is empty. There is an early return path for when the array has fewer than two elements that assumes there is at least one element. If this is not the case, there will be an out-of-bounds access that will cause a runtime error.

```
func PreHandlingElems(flagArray uint32, elems []Byte32) (*Hash, error) {
	ret := make([]*big.Int, len(elems))
	var err error

	for i, elem := range elems {
		if flagArray&(1<<i) != 0 {
			ret[i], err = elem.Hash()
			if err != nil {
				return nil, err
			}
		} else {
			ret[i] = new(big.Int).SetBytes(elem[:])
		}
	}

	if len(ret) < 2 {
		return NewHashFromBigInt(ret[0]), nil
```

Figure 10.1: When `ret` is empty, the array access will cause a runtime error.
(`zktrie/types/util.go#40–57`)

Figure 10.2 shows tests that demonstrate this issue by directly calling `PreHandlingElems` with an empty array and by triggering the issue via the `NodeHash` function.

```
func TestEmptyPreHandlingElems(t *testing.T) {
	flagArray := uint32(0)
	elems := make([]Byte32, 0)

	_, err := PreHandlingElems(flagArray, elems)
	assert.NoError(t, err)
}
```

```
func TestPrehandlingElems(t *testing.T) {
	k := zkt.Byte32{1}
	k_hash_int, _ := k.Hash()
	k_hash := zkt.NewHashFromBigInt(k_hash_int)

	value1 := (&[0]zkt.Byte32{})[:]
	node, _ := NewLeafNode(k_hash, 0, value1).NodeHash()
	t.Log(node)
}
```

Figure 10.2: Tests that trigger the out-of-bounds access

Note that the `TrieUpdate` exported function would also trigger the same issue if called with an empty `vPreimage` argument, but this is checked in the function.

It is also possible to trigger this panic from the Rust API by calling `ZkTrieNode::parse` with the byte array obtained from the `Value()` function operated on a maliciously constructed leaf node. This is because `ZkTrieNode::parse` will eventually call `NewNodeFromBytes` and the `NodeHash` function on that node. The `NewNodeFromBytes` function also does not validate that the newly created node is well formed (TOB-ZKTRIE-14).

```
#[test]
fn test_zktrienode_parse() {
    let buff =
hex::decode("011baa09b39b1016e6be4467f3d58c1e1859d5e883514ff707551a9355a5941e2200000
00000").unwrap();
    let _node = ZkTrieNode::parse(&buff);
}
```

Figure 10.3: A test that triggers the out-of-bounds access from the Rust API

Both the `Data()` and `String()` functions also panic when operated on a `Node` receiver with an empty `ValuePreimage` array:

```
// Data returns the wrapped data inside LeafNode and cast them into bytes
// for other node type it just return nil
func (n *Node) Data() []byte {
	switch n.Type {
	case NodeTypeLeaf:
		var data []byte
		hdata := (*reflect.SliceHeader)(unsafe.Pointer(&data))
		//TODO: uintptr(reflect.ValueOf(n.ValuePreimage).UnsafePointer())
should be more elegant but only available until go 1.18
		hdata.Data = uintptr(unsafe.Pointer(&n.ValuePreimage[0]))
		hdata.Len = 32 * len(n.ValuePreimage)
		hdata.Cap = hdata.Len
		return data
	default:
		return nil
	}
}
```

Figure 10.4: `zktrie/trie/zk_trie_node.go#170–185`

```
// String outputs a string representation of a node (different for each type).
func (n *Node) String() string {
	switch n.Type {
	case NodeTypeParent: // {Type || ChildL || ChildR}
		return fmt.Sprintf("Parent L:%s R:%s", n.ChildL, n.ChildR)
	case NodeTypeLeaf: // {Type || Data...}
		return fmt.Sprintf("Leaf I:%v Items: %d, First:%v", n.NodeKey,
len(n.ValuePreimage), n.ValuePreimage[0])
	case NodeTypeEmpty: // {}
		return "Empty"
	default:
		return "Invalid Node"
	}
}
```

Figure 10.5: `zktrie/trie/zk_trie_node.go#230–242`

Exploit Scenario

An attacker calls the public Rust `ZkTrieNode::parse` function with a maliciously chosen buffer, causing the system to experience a runtime error.

Recommendations

Short term, document which properties need to hold for all data structures. Ensure that edge cases are documented in the type constructors, and add checks to validate that functions do not raise a runtime error.

Long term, add fuzz testing to the public Rust and Go APIs.

11. The hash_external function panics with integers larger than 32 bytes

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ZKTRIE-11
Target: `types/util.go`	

Description

The `hash_external` function will cause a runtime error due to an out-of-bounds access if the input integers are larger than 32 bytes.

```go
func hash_external(inp []*big.Int) (*big.Int, error) {
        if len(inp) != 2 {
                return big.NewInt(0), errors.New("invalid input size")
        }
        a := zkt.ReverseByteOrder(inp[0].Bytes())
        b := zkt.ReverseByteOrder(inp[1].Bytes())

        a = append(a, zeros[0:(32-len(a))]...)
        b = append(b, zeros[0:(32-len(b))]...)
```

Figure 11.1: `zktrie/lib.go#31–39`

Exploit Scenario

An attacker causes the system to call `hash_external` with integers larger than 32 bytes, causing the system to experience a runtime error.

Recommendations

Short term, document the function requirements that the integers need to be less than 32 bytes. If the function is reachable by an adversary, add checks to ensure that the runtime error is not reachable.

Long term, carefully check all indexing operations done on adversary-controlled values with respect to out-of-bounds accessing.

12. Mishandling of cgo.Handles causes runtime errors

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ZKTRIE-12
Target: `lib.go`	

Description

The interaction between the Rust and Go codebases relies on the use of `cgo.Handles`. These handles are a way to encode Go pointers between Go and, in this case, Rust. Handles can be passed back to the Go runtime, which will be able to retrieve the original Go value.

According to the documentation, it is safe to represent an error with the zero value, as this is an invalid handle. However, the implementation should take this into account when retrieving the Go values from the handle, as both the `Value` and `Delete` methods for `Handles` panic on invalid handles.

The codebase contains multiple instances of this behavior. For example, the `NewTrieNode` function will return `0` if it finds an error:

```
// parse raw bytes and create the trie node
//export NewTrieNode
func NewTrieNode(data *C.char, sz C.int) C.uintptr_t {
	bt := C.GoBytes(unsafe.Pointer(data), sz)
	n, err := trie.NewNodeFromBytes(bt)
	if err != nil {
		return 0
	}

	// calculate key for caching
	if _, err := n.NodeHash(); err != nil {
		return 0
	}

	return C.uintptr_t(cgo.NewHandle(n))
}
```

Figure 12.1: `zktrie/lib.go#73–88`

However, neither the Rust API nor the Go API takes these cases into consideration. Looking at the Rust API, the `ZkTrieNode::parse` function will simply save the result from `NewTrieNode` regardless of whether it is a valid or invalid Go handle. Then, calling any other function will cause a runtime error due to the use of an invalid handle. This issue is present in all functions implemented for `ZkTrieNode`: `drop`, `node_hash`, and `value_hash`.

We now precisely describe how it fails in the `drop` function case. After constructing a malformed `ZkTrieNode`, the `drop` function will call `FreeTrieNode` on the invalid handle:

```
impl Drop for ZkTrieNode {
    fn drop(&mut self) {
        unsafe { FreeTrieNode(self.trie_node) };
    }
}
```

Figure 12.2: `zktrie/src/lib.rs#127–131`

This will cause a panic given the direct use of the invalid handle on the `Handle.Delete` function:

```
// free created trie node
//export FreeTrieNode
func FreeTrieNode(p C.uintptr_t) { freeObject(p) }

func freeObject(p C.uintptr_t) {
       h := cgo.Handle(p)
       h.Delete()
}
```

Figure 12.3: `zktrie/lib.go#114–131`

The following test triggers the described issue:

```
#[test]
fn invalid_handle_drop() {
    init_hash_scheme(hash_scheme);
    let _nd = ZkTrieNode::parse(&hex::decode("0001").unwrap());
}
//     running 1 test
// panic: runtime/cgo: misuse of an invalid Handle

// goroutine 17 [running, locked to thread]:
// runtime/cgo.Handle.Delete(...)
//         /opt/homebrew/Cellar/go/1.18.3/libexec/src/runtime/cgo/handle.go:137
// main.freeObject(0x14000060d01?)
//         /zktrie/lib.go:130 +0x5c
// main.FreeTrieNode(...)
//         /zktrie/lib.go:116
```

Figure 12.4: A test case that triggers the finding in the `drop` case

Exploit Scenario

An attacker provides malformed data to `ZkTrieNode::parse`, causing it to contain an invalid Go handle. This subsequently causes the system to crash when one of the `value_hash` or `node_hash` functions is called or eventually when the node variable goes out of scope and the `drop` function is called.

Recommendations

Short term, ensure that invalid handles are not used with `Delete` or `Value`; for this, document the Go exported function requirements, and ensure that Rust checks for this before these functions are called.

Long term, add tests that exercise all return paths for both the Go and Rust libraries.

13. Unnecessary unsafe pointer manipulation in Node.Data()

Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-ZKTRIE-13
Target: `trie/zk_trie_node.go`	

Description

The `Node.Data()` function returns the underlying value of a leaf node as a byte slice (i.e., `[]byte`). Since the `ValuePreimage` field is a slice of `zkt.Byte32`s, returning a value of type `[]byte` requires some form of conversion. The implementation, shown in figure 13.1, uses the `reflect` and `unsafe` packages to manually construct a byte slice that overlaps with `ValuePreimage`.

```
case NodeTypeLeaf:
	var data []byte
	hdata := (*reflect.SliceHeader)(unsafe.Pointer(&data))
	//TODO: uintptr(reflect.ValueOf(n.ValuePreimage).UnsafePointer()) should be
more elegant but only available until go 1.18
	hdata.Data = uintptr(unsafe.Pointer(&n.ValuePreimage[0]))
	hdata.Len = 32 * len(n.ValuePreimage)
	hdata.Cap = hdata.Len
	return data
```

Figure 13.1: Unsafe casting from `[]zkt.Byte32` to `[]byte`
(`trie/zk_trie_node.go#174–181`)

Manual construction of slices and unsafe casting between pointer types are error-prone and potentially very dangerous. This particular case appears to be harmless, but it is unnecessary and can be replaced by allocating a byte buffer and copying `ValuePreimage` into it.

Recommendations

Short term, replace this unsafe cast with code that allocated a byte buffer and then copies `ValuePreimage`, as described above.

Long term, evaluate all uses of unsafe pointer manipulation and replace them with a safe alternative where possible.

14. NewNodeFromBytes does not fully validate its input

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ZKTRIE-14
Target: `trie/zk_trie_node.go`	

Description

The `NewNodeFromBytes` function parses a byte array into a value of type `Node`. It checks several requirements for the `Node` value and returns `nil` and an error value if those checks fail. However, it allows a zero-length value for `ValuePreimage` (which allows TOB-ZKTRIE-10 to be exploited) and ignores extra data at the end of leaf and empty nodes. As shown in figure 14.1, the exact length of the byte array is checked in the case of a branch, but is unchecked for empty nodes and only lower-bounded in the case of a leaf node.

```
case NodeTypeParent:
        if len(b) != 2*zkt.HashByteLen {
                return nil, ErrNodeBytesBadSize
        }
        n.ChildL = zkt.NewHashFromBytes(b[:zkt.HashByteLen])
        n.ChildR = zkt.NewHashFromBytes(b[zkt.HashByteLen : zkt.HashByteLen*2])
case NodeTypeLeaf:
        if len(b) < zkt.HashByteLen+4 {
                return nil, ErrNodeBytesBadSize
        }
        n.NodeKey = zkt.NewHashFromBytes(b[0:zkt.HashByteLen])
        mark := binary.LittleEndian.Uint32(b[zkt.HashByteLen : zkt.HashByteLen+4])
        preimageLen := int(mark & 255)
        n.CompressedFlags = mark >> 8
        n.ValuePreimage = make([]zkt.Byte32, preimageLen)
        curPos := zkt.HashByteLen + 4
        if len(b) < curPos+preimageLen*32+1 {
                return nil, ErrNodeBytesBadSize
        }
…
        if preImageSize != 0 {
                if len(b) < curPos+preImageSize {
                        return nil, ErrNodeBytesBadSize
                }
                n.KeyPreimage = new(zkt.Byte32)
                copy(n.KeyPreimage[:], b[curPos:curPos+preImageSize])
        }
case NodeTypeEmpty:
        break
```

Figure 14.1: `preimageLen` and `len(b)` are not fully checked.
(`trie/zk_trie_node.go#78–111`)

Recommendations
Short term, add checks of the total byte array length and the `preimageLen` field to `NewNodeFromBytes`.

Long term, explicitly document the serialization format for nodes, and add tests for incorrect serialized nodes.

15. init_hash_scheme is not thread-safe

Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-ZKTRIE-15
Target: `src/lib.rs`, `lib.go`, `c.go`, `types/hash.go`	

Description

`zktrie` provides a safe-Rust interface around its Go implementation. Safe Rust statically prevents various memory safety errors, including null pointer dereferences and data races. However, when unsafe Rust is wrapped in a safe interface, the unsafe code must provide any guarantees that safe Rust expects. For more information about writing unsafe Rust, consult The Rustonomicon.

The `init_hash_scheme` function, shown in figure 15.1, calls `InitHashScheme`, which is a `cgo` wrapper for the Go function shown in figure 15.2.

```
pub fn init_hash_scheme(f: HashScheme) {
    unsafe { InitHashScheme(f) }
}
```

Figure 15.1: `src/lib.rs#67–69`

```
// notice the function must use C calling convention
//export InitHashScheme
func InitHashScheme(f unsafe.Pointer) {
	hash_f := C.hashF(f)
	C.init_hash_scheme(hash_f)
	zkt.InitHashScheme(hash_external)
}
```

Figure 15.2: `lib.go#65–71`

`InitHashScheme` calls two other functions: first, a C function called `init_hash_scheme` and second, a second Go function (this time, in the hash module) called `InitHashScheme`.

This second Go function is synchronized with a `sync.Once` object, as shown in figure 15.3.

```
func InitHashScheme(f func([]*big.Int) (*big.Int, error)) {
	setHashScheme.Do(func() {
		hashScheme = f
	})
}
```

Figure 15.3: `types/hash.go#29–33`

However, the C function `init_hash_scheme`, shown in figure 15.4, performs a completely unsynchronized write to the global variable `hash_scheme`, which can lead to a data race.

```
void init_hash_scheme(hashF f){
	hash_scheme = f;
}
```

Figure 15.4: `c.go#13–15`

However, the only potential data race comes from multi-threaded initialization, which contradicts the usage recommendation in the `README`, shown in figure 15.5.

```
We must init the crate with a poseidon hash scheme before any actions:

…

zktrie_util::init_hash_scheme(hash_scheme);
```

Figure 15.5: `README.md#8–24`

Recommendations

Short term, add synchronization to `C.init_hash_scheme`, perhaps by using the same `sync.Once` object as `hash.go`.

Long term, carefully review all interactions between C and Rust, paying special attention to anything mentioned in the "How Safe and Unsafe Interact" section of the Rustonomicon.

16. Safe-Rust ZkMemoryDb interface is not thread-safe	
Severity: **High**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ZKTRIE-16
Target: `lib.go`, `src/lib.rs`, `trie/zk_trie_database.go`	

Description

The Go function `Database.Init`, shown in figure 16.1, is not thread-safe. In particular, if it is called from multiple threads, a data race may occur when writing to the map. In normal usage, that is not a problem; any user of the `Database.Init` function is expected to run the function only during initialization, when synchronization is not required.

```
// Init flush db with batches of k/v without locking
func (db *Database) Init(k, v []byte) {
	db.db[string(k)] = v
}
```

Figure 16.1: `trie/zk_trie_database.go#40–43`

However, this function is called by the safe Rust function `ZkMemoryDb::add_node_bytes` (figure 16.2) via the `cgo` function `InitDbByNode` (figure 16.3):

```
pub fn add_node_bytes(&mut self, data: &[u8]) -> Result<(), ErrString> {
    let ret_ptr = unsafe { InitDbByNode(self.db, data.as_ptr(), data.len() as c_int)
};
    if ret_ptr.is_null() {
        Ok(())
    } else {
        Err(ret_ptr.into())
    }
}
```

Figure 16.2: `src/lib.rs#171–178`

```
// flush db with encoded trie-node bytes
//export InitDbByNode
func InitDbByNode(pDb C.uintptr_t, data *C.uchar, sz C.int) *C.char {
	h := cgo.Handle(pDb)
	db := h.Value().(*trie.Database)

	bt := C.GoBytes(unsafe.Pointer(data), sz)
	n, err := trie.DecodeSMTProof(bt)
	if err != nil {
		return C.CString(err.Error())
	} else if n == nil {
		//skip magic string
		return nil
	}

	hash, err := n.NodeHash()
	if err != nil {
		return C.CString(err.Error())
	}
	db.Init(hash[:], n.CanonicalValue())

	return nil

}
```

Figure 16.3: `lib.go#147–170`

Safe Rust is required to never invoke undefined behavior, such as data races. When wrapping unsafe Rust code, including FFI calls, care must be taken to ensure that safe Rust code cannot invoke undefined behavior through that wrapper. (Refer to the "How Safe and Unsafe Interact" section of the Rustonomicon.) Although `add_node_bytes` takes `&mut self`, and thus cannot be called from more than one thread at once, a second reference to the database can be created in a way that Rust's borrow checker cannot track, by calling `new_trie`. Figures 16.4, 16.5, and 16.6 show the call trace by which a pointer to the `Database` is stored in the `ZkTrieImpl`.

```
pub fn new_trie(&mut self, root: &Hash) -> Option<ZkTrie> {
    let ret = unsafe { NewZkTrie(root.as_ptr(), self.db) };

    if ret.is_null() {
        None
    } else {
        Some(ZkTrie { trie: ret })
    }
}
```

Figure 16.4: `src/lib.rs#181–189`

```
func NewZkTrie(root_c *C.uchar, pDb C.uintptr_t) C.uintptr_t {
	h := cgo.Handle(pDb)
	db := h.Value().(*trie.Database)
	root := C.GoBytes(unsafe.Pointer(root_c), 32)

	zktrie, err := trie.NewZkTrie(*zkt.NewByte32FromBytes(root), db)
	if err != nil {
		return 0
	}

	return C.uintptr_t(cgo.NewHandle(zktrie))
}
```

Figure 16.5: *lib.go#174–185*

```
func NewZkTrieImpl(storage ZktrieDatabase, maxLevels int) (*ZkTrieImpl, error) {
	return NewZkTrieImplWithRoot(storage, &zkt.HashZero, maxLevels)
}

// NewZkTrieImplWithRoot loads a new ZkTrieImpl. If in the storage already exists
one
// will open that one, if not, will create a new one.
func NewZkTrieImplWithRoot(storage ZktrieDatabase, root *zkt.Hash, maxLevels int)
(*ZkTrieImpl, error) {
	mt := ZkTrieImpl{db: storage, maxLevels: maxLevels, writable: true}
	mt.rootHash = root
	if *root != zkt.HashZero {
		_, err := mt.GetNode(mt.rootHash)
		if err != nil {
			return nil, err
		}
	}
	return &mt, nil
}
```

Figure 16.6: *trie/zk_trie_impl.go#56–72*

Then, by calling `add_node_bytes` in one thread and `ZkTrie::root()` or some other method that calls `Database.Get()`, one can trigger a data race from safe Rust.

Exploit Scenario

A Rust-based library consumer uses threads to improve performance. Relying on Rust's type system, they assume that thread safety has been enforced and they run `ZkMemoryDb::add_node_bytes` in a multi-threaded scenario. A data race occurs and the system crashes.

Recommendations

Short term, add synchronization to `Database.Init`, such as by calling `db.lock.Lock()`.

Long term, carefully review all interactions between C and Rust, paying special attention to guidance in the "How Safe and Unsafe Interact" section of the Rustonomicon.

17. Some Node functions return the zero hash instead of errors	
Severity: **Informational**	Difficulty: **N/A**
Type: Error Reporting	Finding ID: TOB-ZKTRIE-17
Target: `lib.go`, `trie/zk_trie_node.go`	

Description

The `Node.NodeHash` and `Node.ValueHash` methods each return the zero hash in cases in which an error return would be more appropriate. In the case of `NodeHash`, all invalid node types return the zero hash, the same hash as an empty node (shown in figure 17.1).

```
		case NodeTypeEmpty: // Zero
			n.nodeHash = &zkt.HashZero
		default:
			n.nodeHash = &zkt.HashZero
		}
	}
	return n.nodeHash, nil
```

Figure 17.1: `trie/zk_trie_node.go#149–155`

In the case of `ValueHash`, non-leaf nodes have a zero value hash, as shown in figure 17.2.

```
func (n *Node) ValueHash() (*zkt.Hash, error) {
	if n.Type != NodeTypeLeaf {
		return &zkt.HashZero, nil
	}
```

Figure 17.2: `trie/zk_trie_node.go#160–163`

In both of these cases, returning an error is more appropriate and prevents potential confusion if client software assumes that the main return value is valid whenever the error returned is `nil`.

Recommendations

Short term, have the functions return an error in these cases instead of the zero hash.

Long term, ensure that exceptional cases lead to non-`nil` error returns rather than default values.

18. get_account can read past the buffer

Severity: **High**	Difficulty: **Medium**
Type: Data Exposure	Finding ID: TOB-ZKTRIE-18
Target: `lib.rs`	

Description

The public `get_account` function assumes that the provided key corresponds to an account key. However, if the function is instead called with a storage key, it will cause an out-of-bounds read that could leak secret information.

In the Rust implementation, leaf nodes can have two types of values: accounts and storage. Account values have a size of either 128 or 160 bytes depending on whether they include one or two code hashes. On the other hand, storage values always have a size of 32 bytes.

The `get_account` function takes a key and returns the account associated with it. In practice, it computes the value pointer associated with the key and reads 128 or 160 bytes at that address.

If the key contains a storage value rather than an account value, then `get_account` reads 96 or 128 bytes past the buffer. This is shown in figure 18.4.

```
// get account data from account trie
pub fn get_account(&self, key: &[u8]) -> Option<AccountData> {
    self.get::<ACCOUNTSIZE>(key).map(|arr| unsafe {
        std::mem::transmute::<[u8; FIELDSIZE * ACCOUNTFIELDS], AccountData>(arr)
    })
}
```

Figure 18.1: `get_account` calls `get` with type `ACCOUNTSIZE` and `key`.
(`zktrie/src/lib.rs#230–235`)

```
// all errors are reduced to "not found"
fn get<const T: usize>(&self, key: &[u8]) -> Option<[u8; T]> {
    let ret = unsafe { TrieGet(self.trie, key.as_ptr(), key.len() as c_int) };

    if ret.is_null() {
        None
    } else {
        Some(must_get_const_bytes::<T>(ret))
    }
}
```

Figure 18.2: get calls must_get_const_bytes with type ACCOUNTSIZE and the pointer returned by TrieGet. (zktrie/src/lib.rs#214–223)

```
fn must_get_const_bytes<const T: usize>(p: *const u8) -> [u8; T] {
    let bytes = unsafe { std::slice::from_raw_parts(p, T) };
    let bytes = bytes
        .try_into()
        .expect("the buf has been set to specified bytes");
    unsafe { FreeBuffer(p.cast()) }
    bytes
}
```

Figure 18.3: must_get_const_bytes calls std::slice::from_raw_parts with type ACCOUNTSIZE and pointer p to read ACCOUNTSIZE bytes from pointer p. (zktrie/src/lib.rs#100–107)

```
#[test]
fn get_account_overflow() {
    let storage_key =

hex::decode("0000000000000000000000000000000000000000000000000000000000000000")
            .unwrap();
    let storage_data = [10u8; 32];

    init_hash_scheme(hash_scheme);
    let mut db = ZkMemoryDb::new();
    let root_hash = Hash::from([0u8; 32]);
    let mut trie = db.new_trie(&root_hash).unwrap();

    trie.update_store(&storage_key, &storage_data).unwrap();

    println!("{:?}", trie.get_account(&storage_key).unwrap());
}

// Sample output (picked from a sample of ten runs):
// [[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10], [160, 113, 63, 0, 2, 0, 0, 0, 161,
67, 240, 40, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 24, 158, 63, 0, 2, 0, 0, 0, 17, 72, 240, 40,
1, 0, 0, 0], [16, 180, 85, 254, 1, 0, 0, 0, 216, 179, 85, 254, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Figure 18.4: This is a proof-of-concept demonstrating the buffer over-read. When run with cargo test get_account_overflow -- --nocapture, it prints 128 bytes with the last 96 bits being over-read.

Exploit Scenario

Suppose the Rust program leaves secret data in memory. An attacker can interact with the zkTrie to read secret data out-of-bounds.

Recommendations

Short term, have `get_account` return an error when it is called on a key containing a storage value. Additionally, this logic should be moved to the Go implementation instead of residing in the Rust bindings.

Long term, review all unsafe code, especially code related to pointer manipulation, to prevent similar issues.

19. Unchecked usize to c_int casts allow hash collisions by length misinterpretation

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ZKTRIE-19
Target: `lib.rs`	

Description

A set of unchecked integer casting operations can lead to hash collisions and runtime errors reached from the public Rust interface.

The Rust library regularly needs to convert the input function's byte array length from the `usize` type to the `c_int` type. Depending on the architecture, these types might differ in size and signedness. This difference allows an attacker to provide an array with a maliciously chosen length that will be cast to a different number. The attacker can choose to manipulate the array and cast the value to a smaller value than the actual array length, allowing the attacker to create two leaf nodes from different byte arrays that result in the same hash. The attacker is also able to cast the value to a negative number, causing a runtime error when the Go library calls the `GoBytes` function.

The issue is caused by the explicit and unchecked cast using the `as` operator and occurs in the `ZkTrieNode::parse`, `ZkMemoryDb::add_node_bytes`, `ZkTrie::get`, `ZkTrie::prove`, `ZkTrie::update`, and `ZkTrie::delete` functions (all of which are public). Figure 19.1 shows `ZkTrieNode::parse`:

```
impl ZkTrieNode {
    pub fn parse(data: &[u8]) -> Self {
        Self {
            trie_node: unsafe { NewTrieNode(data.as_ptr(), data.len() as c_int) },
        }
    }
```

Figure 19.1: `zktrie/src/lib.rs#133–138`

To achieve a collision for nodes constructed from different byte arrays, first observe that `(c_int::MAX as usize) * 2 + 2` is `0` when cast to `c_int`. Thus, creating two nodes that have the same prefix and are then padded with different bytes with that length will cause the Go library to interpret only the common prefix of these nodes. The following test showcases this exploit.

```
#[test]
fn invalid_cast() {
    init_hash_scheme(hash_scheme);
    // common prefix
    let nd =
&hex::decode("012098f5fb9e239eab3ceac3f27b81e481dc3124d55ffed523a839ee8446b648640101
00000000000000000000000000000000000000000000000000000018282256f8b00").unwrap();

    // create node1 with prefix padded by zeroes
    let mut vec_nd = nd.to_vec();
    let mut zero_padd_data = vec![0u8; (c_int::MAX as usize) * 2 + 2];
    vec_nd.append(&mut zero_padd_data);
    let node1 = ZkTrieNode::parse(&vec_nd);

    // create node2 with prefix padded by ones
    let mut vec_nd = nd.to_vec();
    let mut one_padd_data = vec![1u8; (c_int::MAX as usize) * 2 + 2];
    vec_nd.append(&mut one_padd_data);
    let node2 = ZkTrieNode::parse(&vec_nd);

    // create node3 with just the prefix
    let node3 =
ZkTrieNode::parse(&hex::decode("012098f5fb9e239eab3ceac3f27b81e481dc3124d55ffed523a8
39ee8446b64864010100000000000000000000000000000000000000000000000000000018282256f
8b00").unwrap());

    // all hashes are equal
    assert_eq!(node1.node_hash(), node2.node_hash());
    assert_eq!(node1.node_hash(), node3.node_hash());
}
```

Figure 19.2: A test showing three different leaf nodes with colliding hashes

This finding also allows an attacker to cause a runtime error by choosing the data array with the appropriate length that will cause the cast to result in a negative number. Figure 19.2 shows a test that triggers the runtime error for the `parse` function:

```
#[test]
fn invalid_cast() {
    init_hash_scheme(hash_scheme);

    let data = vec![0u8; c_int::MAX as usize + 1];
    println!("{:?}", data.len() as c_int);

    let _nd = ZkTrieNode::parse(&data);
}

// running 1 test
// -2147483648
// panic: runtime error: gobytes: length out of range

// goroutine 17 [running, locked to thread]:
```

```
// main._Cfunc_GoBytes(...)
//         _cgo_gotypes.go:102
// main.NewTrieNode.func1(0x14000062de8?, 0x80000000)
//         /zktrie/lib.go:78 +0x50
// main.NewTrieNode(0x14000062e01?, 0x2680?)
//         /zktrie/lib.go:78 +0x1c
```

Figure 19.3: A test that triggers the issue, whose output shows the reinterpreted length of the array

Exploit Scenario

An attacker provides two different byte arrays that will have the same `node_hash`, breaking the assumption that such nodes are hard to obtain.

Recommendations

Short term, have the code perform the cast in a checked manner by using the `c_int::try_from` method to allow validation if the conversion succeeds. Determine whether the Rust functions should allow arbitrary length inputs; document the length requirements and assumptions.

Long term, regularly run Clippy in pedantic mode to find and fix all potentially dangerous casts.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories

Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.

Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

We identified the following code quality issues through manual and automatic code review.

- **Unnecessary parentheses on function call**

```
// NewSecure creates a trie
// SecureBinaryTrie bypasses all the buffer mechanism in *Database, it
directly uses the
// underlying diskdb
func NewZkTrie(root zkt.Byte32, db ZktrieDatabase) (*ZkTrie, error) {
	maxLevels := NodeKeyValidBytes * 8
	tree, err := NewZkTrieImplWithRoot((db),
zkt.NewHashFromBytes(root.Bytes()), maxLevels)
```

Figure C.1: `zktrie/trie/zk_trie.go#49–54`

- **Typo in code comment**

```
// reduce 2 fieds into one
```

Figure C.2: `types/util.go#L8-L8`

- **Copy-pasted code comment**

```
// ErrNodeKeyAlreadyExists is used when a node key already exists.
ErrInvalidField = errors.New("Key not inside the Finite Field")
```

Figure C.3: `zktrie/trie/zk_trie_impl.go#20–21`

- **Empty `if` branch**

```
if e != nil {
	//fmt.Println("err on NodeTypeEmpty mt.addNode ", e)
}
return r, e
```

Figure C.4: `zktrie/trie/zk_trie_impl.go#190–193`

- **Superfluous `nil` error check**

```
if err != nil {
	return err
}

return nil
```

Figure C.5: `zktrie/trie/zk_trie_impl.go#120–124`

- **Potential panic in function returning `Result`.** Instead, the function should return a `Result` with the appropriate message.

```
pub fn update_account(
    &mut self,
    key: &[u8],
    acc_fields: &AccountData,
) -> Result<(), ErrString> {
    let acc_buf: &[u8; FIELDSIZE * ACCOUNTFIELDS] = unsafe {
        let ptr = acc_fields.as_ptr();
        ptr.cast::<[u8; FIELDSIZE * ACCOUNTFIELDS]>()
            .as_ref()
            .expect("casted ptr can not be null")
```

Figure C.6: `zktrie/src/lib.rs#279–288`

- **Typo in code comment**

```
// ValuePreimage can store at most 256 byte32 as fields (represnted by
BIG-ENDIAN integer)
```

Figure C.7: "represnted" should be "represented"
(`zktrie/trie/zk_trie_node.go#41`*)*

- **Redundant error handling.** The following code should simply check whether `err!= nil` and then return `nil, err`.

```
if err == ErrKeyNotFound {
        return nil, ErrKeyNotFound
} else if err != nil {
        return nil, err
}
```

Figure C.8: `zktrie/trie/zk_trie_impl.go#508–512`

- **Typo in variable name.** The variables should read `FIELD` instead of `FILED`.

```
static FILED_ERROR_READ: &str = "invalid input field";
static FILED_ERROR_OUT: &str = "output field fail";
```

Figure C.9: `zktrie/src/lib.rs#309–310`

- **The difference between `NewHashFromBytes` and `NewHashFromBytesChecked` is not explicitly documented.** `NewHashFromBytes` truncates its input, while `NewHashFromCheckedBytes` returns an error if the input is the wrong length.

```
// NewHashFromBytes returns a *Hash from a byte array considered to be
// a represent of big-endian integer, it swapping the endianness
// in the process.
func NewHashFromBytes(b []byte) *Hash {
	var h Hash
	copy(h[:], ReverseByteOrder(b))
	return &h
}

// NewHashFromCheckedBytes is the intended method to get a *Hash from a byte
array
// that previously has ben generated by the Hash.Bytes() method. so it check
the
// size of bytes to be expected length
func NewHashFromCheckedBytes(b []byte) (*Hash, error) {
	if len(b) != HashByteLen {
		return nil, fmt.Errorf("expected %d bytes, but got %d bytes",
HashByteLen, len(b))
	}
	return NewHashFromBytes(b), nil
}
```

Figure C.10: `types/hash.go#111–128`

- **Unclear comment.** The "create-delete issue" should be documented.

```
//mitigate the create-delete issue: do not delete unexisted key
```

Figure C.11: `trie/zk_trie.go#118`

- **TrieDelete swallows errors.** This error should be propagated to the Rust bindings like other error returns.

```
// delete leaf, silently omit any error
//export TrieDelete
func TrieDelete(p C.uintptr_t, key_c *C.uchar, key_sz C.int) {
	h := cgo.Handle(p)
	tr := h.Value().(*trie.ZkTrie)
	key := C.GoBytes(unsafe.Pointer(key_c), key_sz)
	tr.TryDelete(key)
}
```

Figure C.12: `lib.go#243–250`

D. Automated Analysis Tool Configuration

As part of this assessment, we used the tools described below to perform automated testing of the codebase.

D.1. golangci-lint

We used the static analyzer aggregator golangci-lint to quickly analyze the codebase.

D.2. Semgrep

We used the static analyzer Semgrep to search for weaknesses in the source code repository. Note that these rule sets will output repeated results, which should be ignored.

```
git clone git@github.com:dgryski/semgrep-go.git
git clone https://github.com/0xdea/semgrep-rules

semgrep --metrics=off --sarif --config p/r2c-security-audit
semgrep --metrics=off --sarif --config p/trailofbits
semgrep --metrics=off --sarif --config https://semgrep.dev/p/gosec
semgrep --metrics=off --sarif --config
https://raw.githubusercontent.com/snowflakedb/gosnowflake/master/.semgrep.yml
semgrep --metrics=off --sarif --config semgrep-go
semgrep --metrics=off --sarif --config semgrep-rules
```

Figure D.1: The commands used to run Semgrep

D.3. CodeQL

We analyzed the Go codebase with Trail of Bits's private CodeQL queries. We recommend that Scroll review CodeQL's licensing policies if it intends to run CodeQL.

```
# Create the go database
codeql database create codeql.db --language=go

# Run all go queries
codeql database analyze codeql.db --additional-packs ~/.codeql/codeql-repo
--format=sarif-latest --output=codeql_tob_all.sarif -- tob-go-all
```

Figure D.2: Commands used to run CodeQL

D.4. go cover

We ran the go cover tool with the following command on both the `trie` and `types` folders to obtain a test coverage report:

`go test -cover -coverprofile c.out && go tool cover -html=c.out`

Note that this needs to be run separately for each folder.

D.5. cargo audit

This tool audits `Cargo.lock` against the RustSec advisory database. This tool did not reveal any findings but should be run every time new dependencies are included in the codebase.

D.6. cargo-llvm-cov

cargo-llvm-cov generates Rust code coverage reports. We used the `cargo llvm-cov --open` command to generate a coverage report for the Rust tests.

D.7. Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy --workspace -- -W clippy::pedantic` in the root directory of the project runs the tool. Clippy warns about casting operations from `usize` to `c_int` that resulted in finding TOB-ZKTRIE-19.

E. Go Fuzzing Harness

During the assessment, we wrote a fuzzing harness for the Merkle tree proof verifier function. Because native Go does not support all necessary types, some proof structure types had to be manually constructed. Some parts of the proof were built naively, and the fuzzing harness can be iteratively improved by Scroll's team. Running the `go test -fuzz=FuzzVerifyProofZkTrie` command will start the fuzzing campaign.

```go
// Truncate or pad array to length sz
func padtolen(arr []byte, sz int) []byte {
	if len(arr) < sz {
		arr = append(arr, bytes.Repeat([]byte{0}, sz-len(arr))...)
	} else {
		arr = arr[:sz]
	}
	return arr
}

func FuzzVerifyProofZkTrie(f *testing.F) {
	zkTrie, _ := newZkTrieImpl(NewZkTrieMemoryDb(), 10)

	k := zkt.NewHashFromBytes(bytes.Repeat([]byte("a"), 32))
	vp := []zkt.Byte32{*zkt.NewByte32FromBytes(bytes.Repeat([]byte("b"), 32))}
	node := NewLeafNode(k, 1, vp)

	f.Fuzz(func(t *testing.T, existence bool, depth uint, notempties []byte,
siblings []byte, nodeAuxA []byte, nodeAuxB []byte) {
		notempties = padtolen(notempties, 30)

		typedsiblings := make([]*zkt.Hash, 10)
		for i := range typedsiblings {
			typedsiblings[i] = zkt.NewHashFromBytes(padtolen(siblings, 32))
		}

		typedata := [30]byte{}
		copy(typedata[:], notempties)

		var nodeAux *NodeAux
		if !existence {
			nodeAux = &NodeAux{
				Key:   zkt.NewHashFromBytes(padtolen(nodeAuxA, 32)),
				Value: zkt.NewHashFromBytes(padtolen(nodeAuxB, 32)),
			}
		}

		proof := &Proof{
			Existence:  existence,
			depth:      depth,
			notempties: typedata,
			Siblings:   typedsiblings,
                        NodeAux:    nodeAux,
                }

                if VerifyProofZkTrie(zkTrie.rootHash, proof, node) {
                        panic("valid proof")
                }
        })
}
```

Figure E.1: The fuzzing harness for `VerifyProofZkTrie`

F. Go Randomized Test

During the assessment, we wrote a simple randomized test to verify that the behavior of the zkTrie matches an ideal tree (i.e., an associative array). The test currently fails due to the verifier misbehavior described in this report.

The test can be run with `go test -v -run TestZkTrie_BasicRandomTest`.

```
// Basic randomized test to verify the trie's set, get, and remove features.
func TestZkTrie_BasicRandomTest(t *testing.T) {
	root := zkt.Byte32{}
	db := NewZkTrieMemoryDb()
	zkTrie, err := NewZkTrie(root, db)
	assert.NoError(t, err)

	idealTrie := make(IdealTrie)

	const NUM_BYTES = 32
	const NUM_KEYS = 1024

	keys := random_hex_str_array(t, NUM_BYTES, NUM_KEYS)
	data := random_hex_str_array(t, NUM_BYTES, NUM_KEYS)

	// PHASE 1: toss a coin and set elements (Works)
	for i := 0; i < len(keys); i++ {
		if toss(t) == true {
			set(t, zkTrie, idealTrie, keys[i], data[i])
		}
	}
	// PHASE 2: toss a coin and get elements (Fails)
	for i := 0; i < len(keys); i++ {
		if toss(t) == true {
			get(t, zkTrie, idealTrie, keys[i])
		}
	}
	// PHASE 3: toss a coin and remove elements (Fails)
	for i := 0; i < len(keys); i++ {
		if toss(t) == true {
			remove(t, zkTrie, idealTrie, keys[i])
		}
	}
}
```

Figure F.1: A basic randomized test for ZkTrie (`trie/zk_trie_test.go`)

```
type IdealTrie = map[string][]byte

func random_hex_str(t *testing.T, str_bytes int) string {
	outBytes := make([]byte, str_bytes)
	n, err := rand.Read(outBytes)
	assert.NoError(t, err)
	assert.Equal(t, n, str_bytes)

	return hex.EncodeToString(outBytes)
}

func random_hex_str_array(t *testing.T, str_bytes int, array_size int) []string {
	out := []string{}
	for i := 0; i < array_size; i++ {
		hex_str := random_hex_str(t, str_bytes)
		out = append(out, hex_str)
	}
	return out
}

// Randomly returns true or false
func toss(t *testing.T) bool {
	n, err := rand.Int(rand.Reader, big.NewInt(2))
	assert.NoError(t, err)
	return (n.Cmp(big.NewInt(1)) == 0)
}

func hex_to_bytes(t *testing.T, hex_str string) []byte {
	key, err := hex.DecodeString(hex_str)
	assert.NoError(t, err)
	return key
}

func bytes_to_byte32(b []byte) []zkt.Byte32 {
	if len(b)%32 != 0 {
		panic("unaligned arrays are unsupported")
	}
	l := len(b) / 32
	out := make([]zkt.Byte32, l)

	i := 0
	for i < l {
		copy(out[i][:], b[i*32:(i+1)*32])
		i += 1
	}

	return out
}

// Tests that an inclusion proof for a given key can be generated and verified.
func verify_inclusion(t *testing.T, zkTrie *ZkTrie, key []byte) {
	lvl := 100

	secureKey, err := zkt.ToSecureKey(key)
	assert.NoError(t, err)
	proof, node, err := BuildZkTrieProof(zkTrie.tree.rootHash, secureKey, lvl,
zkTrie.tree.GetNode)
	assert.NoError(t, err)
	valid := VerifyProofZkTrie(zkTrie.tree.rootHash, proof, node)
	assert.True(t, valid)
	hash, err := proof.Verify(node.nodeHash, node.NodeKey)
	assert.NoError(t, err)
	assert.NotNil(t, hash)
}

// Tests that a non-inclusion proof for a given key can be generated and verified.
func verify_noninclusion(t *testing.T, zkTrie *ZkTrie, key []byte) {
	lvl := 100

	secureKey, err := zkt.ToSecureKey(key)
	assert.NoError(t, err)
	proof, node, err := BuildZkTrieProof(zkTrie.tree.rootHash, secureKey, lvl,
zkTrie.tree.GetNode)
	assert.NoError(t, err)
	valid := VerifyProofZkTrie(zkTrie.tree.rootHash, proof, node)
	assert.False(t, valid)
	hash, err := proof.Verify(node.nodeHash, node.NodeKey)
	assert.Error(t, err)
	assert.Nil(t, hash)
}

// Verifies that adding elements from the trie works.
func set(t *testing.T, zkTrie *ZkTrie, idealTrie IdealTrie, key_hex string, data_hex
string) {
	vFlag := uint32(1)
	key := hex_to_bytes(t, key_hex)
	data := hex_to_bytes(t, data_hex)
	err := zkTrie.TryUpdate(key, vFlag, bytes_to_byte32(data))
	assert.NoError(t, err)

	idealTrie[key_hex] = data

	verify_inclusion(t, zkTrie, key)
}

// Verifies that retrieving elements from the trie works.
func get(t *testing.T, zkTrie *ZkTrie, idealTrie IdealTrie, key_hex string) {
	key := hex_to_bytes(t, key_hex)

	ideal_data, ok := idealTrie[key_hex]
	if ok {
		trie_data, err := zkTrie.TryGet(key)
		assert.NoError(t, err)
		assert.True(t, reflect.DeepEqual(trie_data, ideal_data))
		verify_inclusion(t, zkTrie, key)
	} else {
		_, err := zkTrie.TryGet(key)
		assert.Equal(t, ErrKeyNotFound, err)
		verify_noninclusion(t, zkTrie, key)
	}
}
```

```
// Verifies that removing elements from the trie works.
func remove(t *testing.T, zkTrie *ZkTrie, idealTrie IdealTrie, key_hex string) {
	key := hex_to_bytes(t, key_hex)

	_, ok := idealTrie[key_hex]
	if ok {
		delete(idealTrie, key_hex)

		err := zkTrie.TryDelete(key)
		assert.NoError(t, err)
	} else {
		_, err := zkTrie.TryGet(key)
		assert.Equal(t, ErrKeyNotFound, err)
	}

	verify_noninclusion(t, zkTrie, key)
}
```

Figure F.2: Helper functions for `TestZkTrie_BasicRandomTest` (`trie/zk_trie_test.go`)

G. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On September 13, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Scroll team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

Trail of Bits received fix PRs for findings TOB-ZKTRIE-1, TOB-ZKTRIE-4, TOB-ZKTRIE-5, TOB-ZKTRIE-16, TOB-ZKTRIE-18, and TOB-ZKTRIE-19. Scroll did not provide fix PRs for the medium-severity findings TOB-ZKTRIE-2, TOB-ZKTRIE-10, and TOB-ZKTRIE-12; the low-severity finding TOB-ZKTRIE-11; or the remaining findings, all of which are of informational severity.

In summary, of the 19 issues described in this report, Scroll has resolved five issues and has partially resolved one issue. No fix PRs were provided for the remaining 13 issues, so their fix statuses are undetermined. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Lack of domain separation allows proof forgery	Resolved
2	Lack of proof validation causes denial of service on the verifier	Undetermined
3	Two incompatible ways to generate proofs	Undetermined
4	BuildZkTrieProof does not populate NodeAux.Value	Resolved
5	Leaf nodes with different values may have the same hash	Resolved
6	Empty UpdatePreimage function body	Undetermined
7	CanonicalValue is not canonical	Undetermined

8	ToSecureKey and ToSecureKeyBytes implicitly truncate the key	Undetermined
9	Unused key argument on the bridge_prove_write function	Undetermined
10	The PreHandlingElems function panics with an empty elems array	Undetermined
11	The hash_external function panics with integers larger than 32 bytes	Undetermined
12	Mishandling of cgo.Handles causes runtime errors	Undetermined
13	Unnecessary unsafe pointer manipulation in Node.Data()	Undetermined
14	NewNodeFromBytes does not fully validate its input	Undetermined
15	init_hash_scheme is not thread-safe	Undetermined
16	Safe-Rust ZkMemoryDb interface is not thread-safe	Resolved
17	Some Node functions return the zero hash instead of errors	Undetermined
18	get_account can read past the buffer	Partially Resolved
19	Unchecked usize to c_int casts allow hash collisions by length misinterpretation	Resolved

Detailed Fix Review Results

TOB-ZKTRIE-1: Lack of domain separation allows proof forgery
Resolved in PR #11.Domain separation is now performed between leaves, `Byte32` values, and several different types of internal branches. Additionally, sequence-length domain separation has been added to `HashElems` as part of the fixes for finding TOB-ZKTRIE-5. `HashElemsWithDomain` does not add its own domain separation and is potentially error-prone. It appears to be used correctly, but the Scroll team should consider making `HashElemsWithDomain` private to prevent misuse.

TOB-ZKTRIE-2: Lack of proof validation causes denial of service on the verifier
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-3: Two incompatible ways to generate proofs
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-4: BuildZkTrieProof does not populate NodeAux.Value
Resolved in PR #11. The `BuildZkTrieProof` function now calls `ValueHash`, which populates `NodeAux`.

TOB-ZKTRIE-5: Leaf nodes with different values may have the same hash
Resolved in PR #11. `NodeHash` now uses `HandlingElemsAndByte32`, which calls `HashElems`, which adds domain separation based on the length of its input to recursive calls in the internal Merkle tree structure.

TOB-ZKTRIE-6: Empty UpdatePreimage function body
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-7: CanonicalValue is not canonical
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-8: ToSecureKey and ToSecureKeyBytes implicitly truncate the key
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-9: Unused key argument on the bridge_prove_write function
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-10: The PreHandlingElems function panics with an empty elems array
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-11: The hash_external function panics with integers larger than 32 bytes
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-12: Mishandling of cgo.Handles causes runtime errors
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-13: Unnecessary unsafe pointer manipulation in Node.Data()
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-14: NewNodeFromBytes does not fully validate its input
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-15: init_hash_scheme is not thread-safe
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-16: Safe-Rust ZkMemoryDb interface is not thread-safe
Resolved in PR #12. The use of the non-thread-safe function `Database.Init` was replaced with `Put`, which is thread-safe. The Scroll team should consider adding cautionary documentation to `Init` to prevent its misuse in the future.

TOB-ZKTRIE-17: Some Node functions return the zero hash instead of errors
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-ZKTRIE-18: get_account can read past the buffer
Partially resolved in PR #13. The `get_account` function now verifies the slice length. However, the function may still misbehave on platforms where Go's `int` type is 32 bits and if given a value above 2^{31}, which may cause the cast to overflow. This seems extremely unlikely to cause problems in practice. However, adding `assert` calls to `ZkTrie::get` and `TrieGetSize` to restrict the size to be below a reasonable upper bound such as 2^{30} would prevent this issue entirely by causing the program to terminate if the affected code is misused on a problematic platform.

TOB-ZKTRIE-19: Unchecked usize to c_int casts allow hash collisions by length misinterpretation

Resolved in PR #14. This implementation will now crash instead of triggering a bug. The Scroll team should watch for possible denials of service when deploying the software.

H. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.