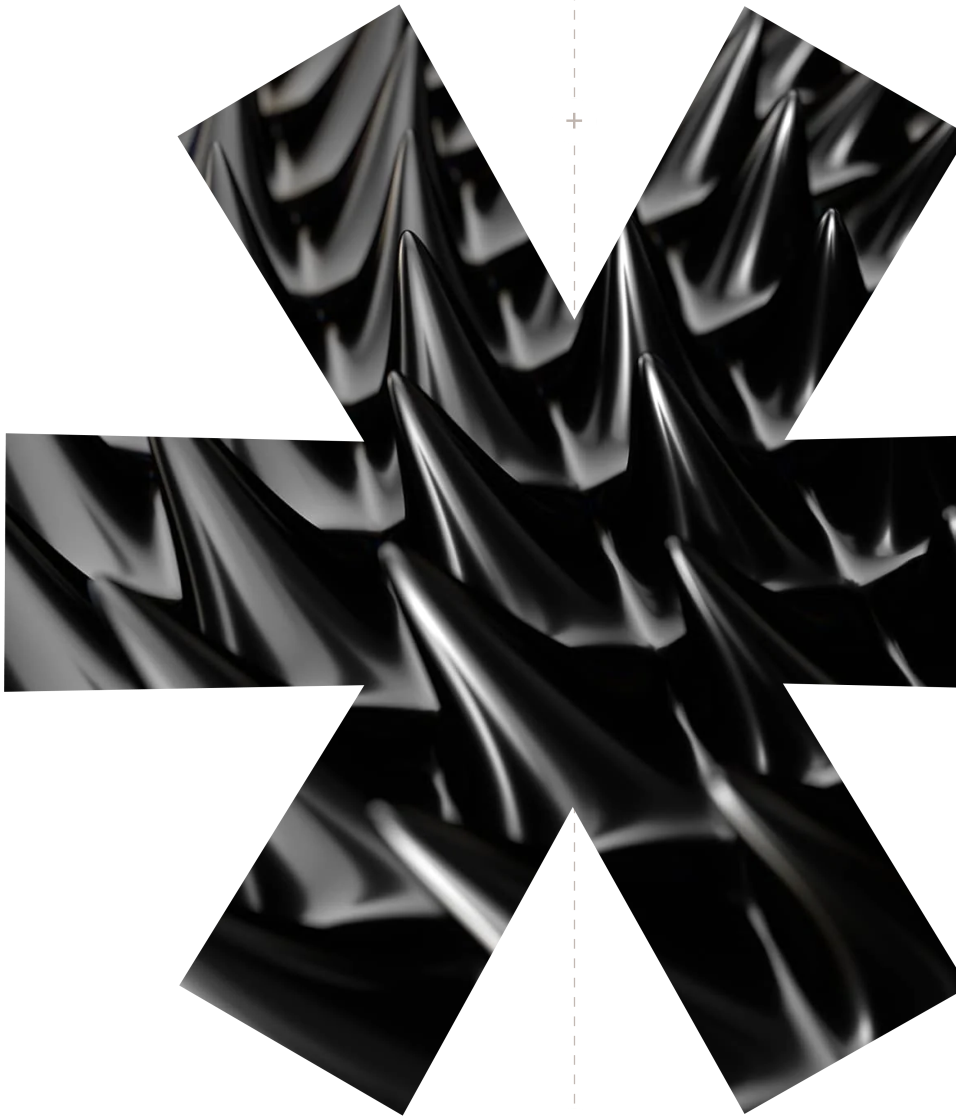
thesisdefense

Security Audit Report

POKT Network

POKT Sparse Merkle Tree

Initial Report // February 20, 2024
Final Report // June 12, 2024



Team Members

Jehad Baeth // Senior Security Auditor
Justin Regele // Senior Security Auditor
Bashir Abu-Amr // Head of Delivery

About Thesis Defense

Thesis Defense serves as the auditing services arm within Thesis, Inc., the venture studio behind tBTC, Fold, Taho, Etcher, and Mezo. Our team of security auditors have carried out hundreds of security audits for decentralized systems across a number of technologies including smart contracts, wallets and browser extensions, bridges, node implementations, cryptographic protocols, and dApps. We offer our services within a variety of ecosystems including Bitcoin, Ethereum + EVMs, Stacks, Cosmos / Cosmos SDK, NEAR and more.

Thesis Defense will employ the Thesis Defense Audit Approach and Audit Process to the in scope service. In the event that certain processes and methodologies are not applicable to the in scope services, we will indicate as such in individual audit or design review SOWs. In addition, Thesis Defense provides clear guidance on successful Security Audit Preparation.

Section_1.0

Scope

Technical Scope

- **Repository:** https://github.com/pokt-network/smt
- **Audit Commit:** 868237978c0b3c0e2added161b36eeb7a3dc93b0
- **Verification Commit:** 3981639bd08cf52a7668c3681cbe2243d957e4ee

Any third-party or dependency library code is considered out of scope, unless explicitly specified as in scope above.

Documentation

- Technical documentation: https://github.com/pokt-network/smt/tree/main/docs

Bibligraphy

- M. Al-Bassam, A. Sonnino, and V. Buterin, “Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities,” arXiv preprint arXiv:1809.09044, vol. 160, 2018.
- M. Al-Bassam, “Lazyledger: A distributed data availability ledger with client-side smart contracts,” arXiv preprint arXiv:1905.09274, 2019.
- R. Dahlberg, T. Pulls, and R. Peeters, “Efficient Sparse Merkle Trees: Caching strategies and secure (non-) membership proofs,” in Secure IT Systems: 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016. Proceedings 21, 2016, pp. 199–215.
- Z. Gao, Y. Hu, and Q. Wu, “Jellyfish Merkle tree,” 2021.
- Z. Amsden et al., “The Libra Blockchain - White Paper,” Whitepaper, 2019.
- D. Olshansky and R. R. Colmeiro, “Relay Mining: Verifiable Multi-Tenant Distributed Rate Limiting,” arXiv preprint arXiv:2305.10672, 2023.
- J. Kalidhindi, A. Kazorian, A. Khera, and C. Pari, “Angela: A sparse, distributed, and highly concurrent merkle tree,” UC Berkeley, Berkeley, 2018.

Table of Contents

Executive Summary

Schedule

This security audit was conducted from February 7, 2024 to February 20, 2024 by 2 security auditors for a total of 4 person-weeks.

Overview

Thesis Defense conducted a manual code review of POKT Network's Sparse Merkle Trie (Trie) implementation, an authenticated key-value data structure.

The POKT Network team has adapted a Sparse Merkle Tree for POKT Network's use case. POKT Network nodes provide applications with RPC relay services. Upon claiming rewards, the node must prove that it provided the work of the service by generating a valid inclusion "closest proof" of a randomly selected branch within the agreed upon root trie.

The SMT algorithm is well tested and mature. The scope of this audit included changes that have been made to improve the efficiency of the algorithm.

Threat Model

For this audit, our team considered a threat model whereby a malicious node is able to generate a valid proof of inclusion to earn illegitimate rewards. We also considered an honest node whose valid proof of inclusion is rejected.

Security by Design

The POKT SMT is designed to efficiently handle sparse data where most of the tree's leaves are empty, and is particularly useful for key value stores, allowing for an efficient proof of non-inclusion. Each leaf corresponds to a key value pair, with that leaf's position predetermined by the hash of that leaf's key.

ProveClosest

We investigated the `ProveClosest` algorithm and found that it is a more efficient version of a non-inclusion proof. `ProveClosest` is a more efficient and specific proof that shows the non-inclusion of a key by demonstrating the emptiness of the closest leaf node, while a normal non-inclusion proof is a more general proof that demonstrates the non-inclusion of a key by showing that no path in the tree leads to that key. Closest proof makes use of the fact that in a SMT, the vast majority of nodes are empty or hold nil values.

Secure Implementation

We found the code to be well organized and correctly implemented, in accordance with security best practices. We investigated the security of the implementation of the POKT SMT in several key areas.

False Sum Proof Protection

We investigated whether a Sum Proof could be forged by manipulating the weight appended to the end of the leaf digest. We concluded that even if an attacker forged a leaf where a weight was hashed into the leaf digest, but a different weight was appended, the proof would fail in the `validateBasic` validation checks.

```
 siblingHash := hashPreimage(spec, proof.SiblingData)
 if eq := bytes.Equal(proof.SideNodes[0], siblingHash); !eq {
  return fmt.Errorf("invalid sibling data hash: got %x but want %x", siblingHash,
proof.SideNodes[0])
 }
```

Here, `proof.SiblingData` is the sibling leaf node of the node being proven. An attacker can add arbitrary data to this value in the Proof, but this verification check will hash the entire data, and in the case of a Sum Proof, will append the weight at the end of the hash, even though the weight has been included in the preimage.

This essentially is enforcement for the weights inside the proof. This is effective because we can provide arbitrary `SiblingData` in the proof and we can have the hash of weight A in the `SiblingData` with weight B appended, but when `SiblingData` is hashed with weight B, then it will not equal the `SideNode` hash. If we leave weight as is in `SiblingData`, the hashes match, but not the weights, resulting in the following error:

```
 invalid sibling data hash: got
ba36043b00bccd6730e474139979b2954573b5e3520581e5cb542
010163f81d3000000000000000a
```

but want

```
 ba36043b00bccd6730e474139979b2954573b5e3520581e5cb542
010163f81d30000000000000005"}
```

Note that the hashes themselves are identical, but the weights differ after being manipulated in proof construction.

Path Representation

Representing the path in a Merkle tree can be done either as a series of bits indicating the node's location based on its position in the tree or as an index value. This method is straightforward and can be implemented efficiently, especially for binary Merkle trees. The binary representation of the index directly corresponds to the path from the root to the leaf node. Currently, Paths in the SMT implementation are represented as a series of bits.

It also can be represented with `N(level,depth)` since depth of all leaf nodes is the same given a specific hash function a path can be represented with depth only. Using an index value to represent the path can be more intuitive and easier to understand, as it directly maps to the position of the leaf node in the tree. An index value is also more compact than a series of bits, as it does not require storing the entire path.

That being said, we think further exploration of the design decision trade-offs between efficiency, compactness, and ease of implementation could have positive impacts for future iterations.

Parallelism in SMT

The current implementation of SMT does not account for parallelism. Literature concerning other implementations of an SMT utilized a variety of techniques such as parallel processing, atomic update operations, and batch processing for further optimizations. We did not find any indication that the code anticipates handling parallel processing.

We investigated the access patterns of existing byte arrays that hold different data used by the code. Optimally, access to such resources should be synchronized when there are multiple writers interacting with it at the same time. We have only been able to identify cases where a single writer interacts with those variables at a time.

After further discussions with the POKT team, it was clarified that each instance of a Trie will be self-hosted by a relay-miner, and therefore accounting for parallelism is not necessary. Concurrent writes to the Trie will not ever be a requirement so long as each relayer is the only one writing to its Trie.

Protection Against Second Preimage Attacks and Shortened Proof Attacks

A second preimage attack on a SMT occurs when an attacker can find two different inputs that produce the same hash output. In the context of Merkle trees, this could mean finding two different sets of data that, when hashed into the tree, result in the same root hash. Currently, the SMT uses node-type specific prefixes (0 for leaf nodes, 1 for inner nodes, and 2 for extension nodes) to protect against second preimage attacks, which is a valid strategy commonly utilized in SMT implementations.

Alternatively, shortened proof attacks are a type of security vulnerability that can occur in Merkle trees when a proof is not constructed correctly or when there is a lack of domain separation between leaf nodes and internal nodes. This type of attack is similar to a second preimage attack, where an attacker can create a valid proof for a data element that is not actually in the original key-value store. Using a different hash function for hashing leaves than for hashing internal nodes helps mitigate the risk of both attacks. We recommend the POKT Network team explore utilizing such a strategy to further enforce domain separation, which provides protection against both second preimage and shortened proof attacks.

Use of Dependencies

We ran dependency analysis tools and did not identify any issues in the use of dependencies.

Tests

There are tests implemented in the repository with benchmark and stress testing documented. These tests are helpful in understanding the intended functionality, and are generally sufficient, but narrow in scope.

Before deploying the new SMT implementation onto mainnet, we recommend running a shadow fork where developers can test the new implementation in a controlled, safe, and isolated environment. This will reduce the risk of disrupting the mainnet or causing unintended consequences, in addition to identifying any issues or bugs that may arise. Furthermore, running a shadow fork would allow conducting a more thorough performance evaluation under different conditions and loads. Lastly, shadow forks can help identify compatibility issues between the new SMT implementation and other system components.

Project Documentation

There is comprehensive documentation available for the currently deployed POKT Network “Morse”. In addition, there is sufficient documentation detailing the research, reasoning, and rationale for the design choices made regarding POKT Network’s protocol “Shannon” upgrade. The code is well commented and the documentation available in the code repository is accurate and helpful.

In addition, we received updates to the technical documentation during the audit. Although we were not able to review the updated documentation thoroughly due to time constraints, the work on this documentation should continue, and we recommend a thorough review of the updated documentation during the verification phase.

Key Findings Table

Issues	Severity	Status
ISSUE #1 Insufficiently Secure Hash Function Can Be Initialized	Low	✓ Fixed
ISSUE #2 Writeable KV Store Could Prove Non-Existent Nodes	Low	✓ Fixed
ISSUE #3 Improve Calling Convention of `VerifyClosestProof` and `verifyProofWithUpdates`	None	✓ Fixed
ISSUE #4 Refactor the Insertion of an Extension Node	None	✓ Fixed

Severity definitions can be found in Appendix A

Section_4.0

Findings

We describe the security issues identified during the security audit, along with their potential impact. We also note areas for improvement and optimizations in accordance with best practices. This includes recommendations to mitigate or remediate the issues we identify, in addition to their status before and after the fix verification.

ISSUE#1

Insufficiently Secure Hash Function Can Be Initialized

✓ Low   ✓ Fixed

Location

smt.go#L74C9-L74C18

Description

In Go, the hash package provides a standard interface for hash functions and checksum algorithms. The `hash.Hash` interface is implemented by all hash functions in the standard library. A hash function suitable for the Merkle tree implementation must satisfy some criteria such as:

- Collision Resistance: For a Merkle tree, a hash function that is collision-resistant is required. This means that it is computationally infeasible to find two different inputs that hash to the same output.
- Preimage Resistance: A hash function that is preimage-resistant means that it is difficult to find the original input given only the hash output. This property is crucial for the security of a Merkle tree, as it prevents an attacker from modifying the data without being detected.
- Efficiency: The hash function should be efficient, as it will be used to compute the hash of many nodes in the Merkle tree.

SMT poses no hardcoded restriction on the type of hash functions that can be used other than that it should implement golang hash interface. Nor the documentation provides guidance on which hash functions can be used.

Impact

The implementation of an insufficiently collision or preimage resistant hash function could undermine the security assumptions of the SMT.

Recommendation

We recommend implementing a code level hard restriction on what hash functions are allowed to initialize the SMT based on collision, preimage attack, and time and space efficiency.

Verification Status

The POKT SMT documentation has been updated to explicitly explain the criteria of a hash function selection.

ISSUE#2

Writeable KV Store Could Prove Non-Existent Nodes

Low ✓ Fixed

Location

kvstore/interfaces.go#L12

Description

A Malicious Prover with write access to the Key Value (KV) store of a Verifier could write arbitrary data over a Leaf Node in order to convince the SMT that the node is an Internal Node, with the intention of writing Proofs for nodes that do not exist. This attack would only work if the Verifier updated the sibling leaf node of the corrupted data, which would update the Merkle Root to reflect the presence of the malicious internal node, allowing the attacker to prove the existence of child nodes of the internal node, without them existing in the SMT.

Impact

The integrity of the SMT's Proving mechanism can be compromised.

Recommendation

We recommend ensuring that the KV store is not writeable externally. The POKT Network team confirmed that the KV store is not replicated.

Verification Status

The POKT SMT documentation explicitly warns users to ensure usage of an externally writable key-value store in production environments.

ISSUE#3

Improve Calling Convention of `VerifyClosestProof` and `verifyProofWithUpdates`

None ✓ Fixed

Location

proofs.go#L320, L323

Description

The way the `verifyProofWithUpdates` looks up Paths by hashing the Key creates an awkward programming interface with the `VerifyClosestProof` algorithm, because a `ClosestProof` provides a Path as a Key. If a new Spec of a `nilPathHasher` is not provided during Proof Verification, double hashing of the Path will occur and the Proof will fail.

Impact

None – no security impact

Recommendation

We recommend revising the calling conventions around `VerifyClosestProof` and `verifyProofWithUpdates` such that the API is not implemented as a workaround.

Verification Status

The updated code encapsulates the creation of the `nilPathHasher` so that the calling convention remains standard across the application.

ISSUE#4

Refactor the Insertion of an Extension Node

None ✓ Fixed

Location

smt.go#L179-L189

Description

The insertion of Extension Nodes uses the same pointer to assign values to Extension Nodes as well as its children. We found this coding pattern confusing.

Impact

None – no security impact.

Recommendation

We recommend updating the Extension Node's child with an explicit variable as in the example below, leaving the pointer variable to only dereference the Node at the current depth.

```
 var extension_node_child = nil
if getPathBit(path, prefixlen) == left {
    extension_node_child = &innerNode{leftChild: newLeaf, rightChild: leaf}
} else {
    extension_node_child = &innerNode{leftChild: leaf, rightChild: newLeaf}
}
ext := extensionNode{child: extension_node_child, path: path, pathBounds:
[2]byte{byte(depth), byte(prefixlen)}}
*last = &ext
```

Verification Status

The POKT Network team has altered the mentioned coding pattern and currently employs separate pointers, specifically for the Child nodes.

Section_5.0

Appendix A

Severity Rating Definitions

At Thesis Defense, we utilize the Immunefi Vulnerability Severity Classification System - v2.3.

Severity	Definition
Critical	• Network not being able to confirm new transactions (total network shutdown) • Unintended permanent chain split requiring hard fork (network partition requiring hard fork) • Direct loss of funds • Permanent freezing of funds (fix requires hardfork)
High	• Unintended chain split (network partition) • Temporary freezing of network transactions by delaying one block by 500% or more of the average block time of the preceding 24 hours beyond standard difficulty adjustments • Causing network processing nodes to process transactions from the mempool beyond set parameters • RPC API crash affecting projects with greater than or equal to 25% of the market capitalization on top of the respective layer
Medium	• Increasing network processing node resource consumption by at least 30% without brute force actions, compared to the preceding 24 hours • Shutdown of greater than or equal to 30% of network processing nodes without brute force actions, but does not shut down the network • Griefing (e.g. no profit motive for an attacker, but damage to the users or the protocol) • A bug in the respective layer 0/1/2 network code that results in unintended smart contract behavior with no concrete funds at direct risk
Low	• Shutdown of greater than 10% or equal to but less than 30% of network processing nodes without brute force actions, but does not shut down the network • Modification of transaction fees outside of design parameters
None	• We make note of issues of no severity that reflect best practice recommendations or opportunities for optimization, including, but not limited to, gas optimization, the divergence from standard coding practices, code readability issues, the incorrect use of dependencies, insufficient test coverage, or the absence of documentation or code comments.

Section_6.0

Appendix B

Thesis Defense Disclaimer

Section_7.0

Appendix C - Discussion of SHA256 Trie

April 9 POKT Question:

`In a SHA256 trie, any update would take 256 steps, while generating a proof could use up to 256 steps.` Can you explain where the 256 steps came from?

May 3 Defense response:

Sparse merkle trees have a fixed depth, assuming no structure-changing optimizations are made at a later step. This is a key characteristic that differentiates them from regular Merkle trees. A fixed depth allows for efficient proofs of membership or non-membership. The proof size only depends on the depth, not the total number of leaves (which can be very large in sparse trees). In addition, the structure of the tree becomes predictable. Since the depth is fixed, the location of any leaf node can be determined based on its index and the underlying hash function.

In expectation, the depth of any given item is O(log n), where n is the number of items in the tree.

(non-)membership proofs are expected to require space O(log n). In the worst case they are bounded by the fixed depth of the SMT, and verification time is always O(1). JMT paper section 4.3.

It is important to note we state that the proof could use up to 256 steps because that is the depth of the trie. Verification would only require as many steps as there were side nodes. So in the cases of sparsely populated tries, less than 256 side nodes is likely.

Here are some resources that discuss this concept in more detail:

- Sparse Merkle Trees: Definitions and Space-Time Trade-Offs with Applications for Balloon
- Achieving Blockchain Scalability with Sparse Merkle Trees and Bloom Filters

May 10 POKT response:

`Sparse merkle trees have a fixed depth, assuming no structure-changing optimizations are made at a later step.` A NIT/clarification I wanted to understand:

1. Is it a `fixed depth` or a `max fixed depth`. I'm not sure if it's terminology or technical details.

The extension node should behave similar to branch nodes in Ethereum's MPT, so log2(256) should be the "max" depth from how it was originally designed. If this is not the case, we may need to go back and re-evaluate.

May 28 Defense response:

We think this could best be understood by thinking of the SMT data structure as theoretical, as opposed to concrete. All nodes will have a fixed depth of 256 because that is the amount of bits in their path (since the hash function in this implementation outputs 256 bits). So in a trie with only one node, you could say that the depth is still 256 because that node's full path is still 256. However, because there is only one node, the algorithm itself will not need to travel all 256 bits before it finds the node, because only 1 bit is needed to find it. So in sparsely populated tries, this implementation we reviewed won't necessarily need to traverse to the full 256 depth to find the node, except in cases where the target node shares 255 bits with another node. The implementation searches for nodes until there are no other nodes that have common prefixes. So while the depth of the trie is actually 256, because of how the nodes are stored in memory, traversing all 256 levels is not always going to happen.

The `smt.depth` function will always return the bit length of whatever hash function is specified for the `PathHasher` in the config:

```
func (spec *TrieSpec) depth() int { return spec.ph.PathSize() * 8 }
```

This is the depth of the trie itself. But there is another depth variable that is used when walking a trie as seen in `Get` and `Prove` functions, which tracks the depth traversed along the in-memory trie data structure until the target node is found. So the depth of any nodes address is going to be 256, but the depth the algorithm needs to travel is going to be less than that, and this is the optimization that Sparse Merkle Trees provides.

In other words, the trie has a depth of 256, but a proof might have a smaller depth because it is only dependent on the common prefix bits of node compared to other siblings in the trie. The `TestSMT_TrieMaxHeightCase` unit test in your repo provides a good example of this. In this test case, two separate keys have common prefix bits up to 254, leaving only the least significant bit as different. This proof would require 256 steps, but if we shortened the common prefix bits, this would also shorten the proof (assuming there are no other nodes with common prefix bits in the trie at that time).

June 6 POKT response:

Got it. Appreciate the explanation.