Security Audit Report

Pocket Network Foundation:

POKT Network Sparse Merkle Trie (SMT)

Initial Audit Report: February 20, 2024

thesisdefense

defense@thesis.co

Table of Contents

About Thesis Defense

Thesis Defense serves as the auditing services arm within Thesis, Inc., the venture studio behind tBTC, Fold, Taho, Etcher, and Embody. Our team of senior security and cryptography auditors has extensive security experience in the decentralized technology space. In addition, the Thesis Defense team has a demonstrated track record in a variety of languages and technologies, including, but not limited to, smart contracts, cryptographic protocols including zk-cryptography, dApps including wallets and browser extensions, and bridges. Thesis Defense has extensive experience conducting security audits across a number of ecosystems, including, but not limited to, Ethereum, Zcash, Stacks, Mina, Polygon, Filecoin, and Bitcoin.

Thesis Defense will employ the Thesis Defense Audit Approach and Audit Process to the above in-scope service. In the event that certain processes and methodologies are not applicable to the aforementioned 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.

Scope

Overview

Thesis Defense conducted a manual code review of POKT Network's Sparse Merkle Trie (Trie) implementation.

Project Team

- **Jehad Baeth**, Senior Security Auditor
- **Justin Regele**, Senior Security Auditor
- **Bashir Abu-Amr**, Senior Technical Writer

Schedule

- **Code Review**: February 7 - 20, 2024
- **Audit Report Delivery**: February 20, 2024

Code

- **Repository**: https://github.com/pokt-network/smt
- **Branch**: `main`
- **Hash**: `868237978c0b3c0e2added161b36eeb7a3dc93b0`

Project Documentation

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

Bibliography

- 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.

Findings

POKT Network's Sparse Merkle Trie (SMT) is 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 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.

In a SHA256 trie, any update would take 256 steps While generating a proof could use up to 256 steps.

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. We investigated the security of the implementation of the POKT SMT in several 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. However, we believe that the SMT implementation can be encapsulated with goroutines for protection against unexpected concurrency edge cases.

Protection against 2nd 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 2nd preimage attacks which is a valid strategy commonly utilized in SMT implementations.
On the other hand, 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 2nd 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. We recommend running a shadow fork of mainnet for testing (Suggestion 2).

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.

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.

Issues and Suggestions

Issues	Status
Issue A: Insufficiently Secure Hash Function Can Be Initialized	Reported
Issue B: Writeable KV Store Could Prove Non-Existent Nodes	Reported

Suggestions	Status
Suggestion 1: Improve Calling Convention of VerifyClosestProof and verifyProofWithUpdates	Reported
Suggestion 2: Test the Implementation On Shadow Fork of Mainnet	Reported
Suggestion 3: Refactor the Insertion of an Extension Node	Reported

Issues

Issue A: Insufficiently Secure Hash Function Can Be Initialized

Location

https://github.com/pokt-network/smt/blob/868237978c0b3c0e2added161b36eeb7a3dc93b0/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 criterias such as:

- Collision Resistance: For a Merkle tree, especially in a cryptographic, you need a hash function that is collision-resistant. This means that it is computationally infeasible to find two different inputs that hash to the same output.
- Preimage Resistance: You also need a hash function that is preimage-resistant, meaning 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.

Remediation

We recommend 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.

Issue B: Writeable KV Store Could Prove Non-Existent Nodes

Location

https://github.com/pokt-network/smt/blob/868237978c0b3c0e2added161b36eeb7a3dc93b0/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.

Remediation

Ensure the KV store is not writeable externally. The POKT network team confirmed that the KV store is not replicated.

Suggestions

Suggestion 1: Improve Calling Convention of VerifyClosestProof and verifyProofWithUpdates

Location

https://github.com/pokt-network/smt/blob/868237978c0b3c0e2added161b36eeb7a3dc93b0/proofs.go#L320
https://github.com/pokt-network/smt/blob/868237978c0b3c0e2added161b36eeb7a3dc93b0/proofs.go#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.

Remediation

We recommend that the calling conventions around `VerifyClosestProof` and `verifyProofWithUpdates` are revised so that the API is not implemented as a workaround.

Suggestion 2: Test the Implementation On a Shadow Fork of Mainnet

Description

Before plugging the new SMT implementation into mainnet, we recommend running a shadow fork where developers can test the new implementation in a controlled, safe, and isolated environment reducing the risk of disrupting the mainnet or causing unintended consequences and identify any issues or bugs that may arise. This would also 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 components.

Suggestion 3: Refactor the Insertion of an Extension Node

Location

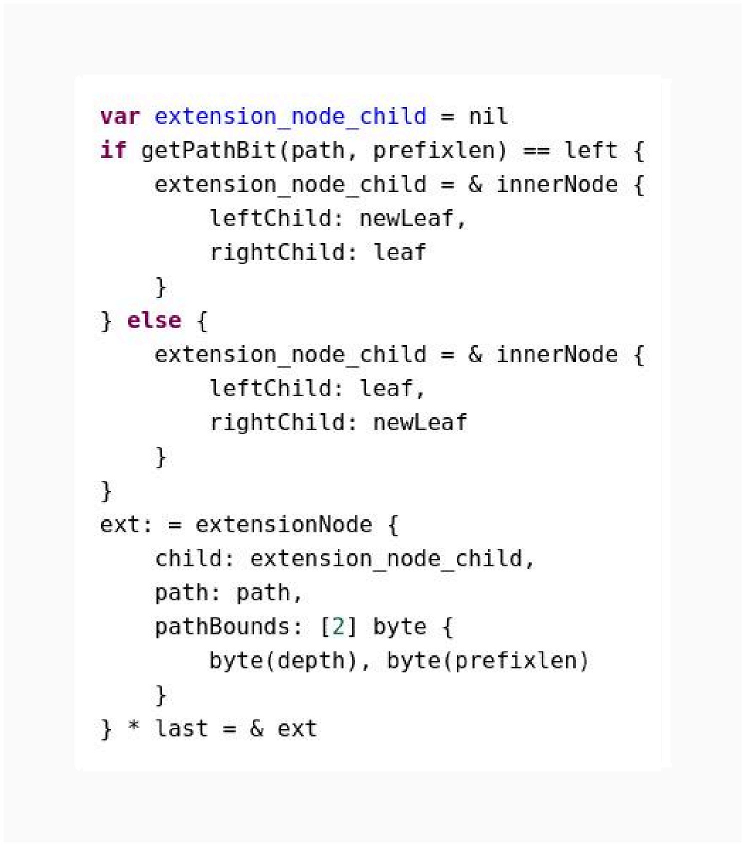
https://github.com/pokt-network/smt/blob/868237978c0b3c0e2added161b36eeb7a3dc93b0/smt.go#L179-L189

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.

Remediation

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
```