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

### Question 2: Can you explain where the 256 steps came from?

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 (http://www.diva-portal.org/smash/get/diva2:936353/FULLTEXT02.pdf)
- Achieving Blockchain Scalability with Sparse Merkle Trees and Bloom Filters (https://medium.com/@kelvinfichter/whats-a-sparse-merkle-tree-acda70aeb837)

---

> We investigated the ProveClosest algorithm and found that it is a more efficient version of a non-inclusion proof.

### Question 3: This is intended to be an inclusion proof of the closest non-empty leaf. Is this not the case from your evaluation?

It is. Our statement was in regard to the efficiency factor. With a sparse (mostly unpopulated) tree, efficiency comes from the fact that non-inclusion (when it happens) are very efficient.

We've reviewed it further, and we see how the use case for the Closest Proof algorithm is part of a challenge/response is to be used. Because the trie is sparse, an arbitrary hash path is unlikely to be populated. When a verifier presents the prover with a challenge, the prover needs to be able to prove the existence of the arbitrary node, and the Prove Closest algorithm is an efficient way to do this.

---

> 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.

### Question 4: Did you validate and confirm that this is indeed the case?

No, this is not the case with the current implementation. We are here speculating on other data representations that could be implemented. This is a comment on the design of the SMT algorithm in general rather than a commentary on the implementation.

---

> 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.

### Question 5: I don't fully understand the suggestion given that we are aiming to build a Trie. Do you have a reference implementation of another Trie that does this?

- The Aztec network SMT implementation uses indexes (https://docs.aztec.network/learn/concepts/storage/trees/indexed_merkle_tree)
- (https://eprint.iacr.org/2021/1263.pdf)

In summary, while the concept of using an index value instead of a path for a leaf node in an SMT is theoretically possible, we mentioned it in the report because it could offer certain optimizations. But it would also require some fundamental changes to the implementation.

Because of the way the implementation revolves around path hashes, we did not suggest this approach be taken. Our comment was on the design choices, and felt it worth mentioning in case it could spark ideas for the development team.

---

> 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

### Question 6: How do other implementation handle parallel updates? Provided that we need to rehash on every insertion, I believe this requires a global lock.

We're not sure how the trie will be hosted, and how often it will be written to and by whom. Does every relayer host their own trie, or are the tries self hosted by every verifier and prover? Investigating how other implementations handle parallel updates is a big task that is beyond the scope of this initial audit, but the following papers might provide better insight:

- (https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F18/projects/reports/project1_report_ver3.pdf)
- (https://arxiv.org/pdf/2311.17441)

---

> However, we believe that the SMT implementation can be encapsulated with goroutines for protection against unexpected concurrency edge cases.

### Question 7: I can understand the need for mutexes to solve this but how would goroutines help?

Correct. While it's correct that Goroutines do not inherently provide thread safety, encapsulating mutexes in goroutines is a commonly used practice which is what we were reffering to. Goroutines provide the concurrency, and mutexes ensure safe access to shared resources. It helps in managing the complexity of concurrent programs and makes the code more robust and easier to maintain.

It’s necessary to keep in mind that the commentary provided about parallelism is discussing a hypothetical case. Real life implementations will definitely have more nuance.

---

> Optimally, access to such resources should be synchronized when there are multiple writers interacting with it at the same time.

In the example where badger is used as the backing key-value store engine, `store.db.Update` handles transactional operations behind the scenes.

### Question 8: Does this account for issues with multiple writers?

Comments in the Badger source code regarding the Update() function indicate that while transactions can be run concurrently, they are not thread safe so it is recommended to run them serially:

> Running transactions concurrently is OK. However, a transaction itself isn't thread safe, and should only be run serially. It doesn't matter if a transaction is created by one goroutine and passed down to other, as long as the Txn APIs are called serially.

- (https://github.com/dgraph-io/badger/blob/main/txn.go#L756C1-L758C70)

For SMT, if there are multiple writers, it would be crucial to ensure that no race conditions occur before implementing parallel writes.

---

> Issue B: Writeable KV Store Could Prove Non-Existent Nodes [...] We recommend ensuring that the KV store is not writeable externally. The POKT Network team confirmed that the KV store is not replicated.

### Question 9: Would this sort of attack also apply to any other Blockchain that uses Merkle Trees? If so, is this just a matter of “secure DevOps practices”?

This is a matter of how the backend database is hosted and accessed. Since it is a Web 2 datastore and not an onchain smart contract, ensuring the security of the datastore would fall in the domain of secure DevOps.