Reviewing Audit Report

240404_Thesis.Defense.-.Pocket.Network.Foundation.-.SMT.-.Security.Audit.Report.Final.pdf

Thesis.Defense-Pocket.Network.Foundation-SMT-Security.Audit.Report.pdf

Threat Model

1. Malicious node is able to generate a valid proof of inclusion
2. Honest node whose proof of inclusion is rejected

Questions / Asks

> **Repository:** https://github.com/pokt-network/smt
>
> **Hash:** 3981639bd08cf52a7668c3681cbe2243d957e4ee

1. `Question`: This hash will need to be updated after we discuss and update `main`.

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

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

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

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

`Extra context`: I recently put together is available at claim_and_proof_lifecycle#merkle-proof-selection. From the godoc of `ClosestProof`:

```
// ProveClosest generates a SparseMerkleProof of inclusion for t
// key with the most common bits as the path provided.
//
// This method will follow the path provided until it hits a lea
// exit. If the leaf is along the path it will produce an inclus
// the key (and return the key-value internal pair) as they shar
// prefix. If however, during the trie traversal according to th
// node is encountered, the traversal backsteps and flips the pa
// depth (ie tries left if it tried right and vice versa). This
// a proof of inclusion is found that has the most common bits w
// provided, biased to the longest common prefix.
```

> 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.

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

`Extra Context`: The goal of an `extensionNode` is to change the depth of the leaf and save size on empty subtrees. This is something we adapted from Ethereum's MPT and Facebook's JMT. I've been meaning to build a visualizer but haven't had time.

```
type extensionNode struct {
    // The path (starting at the root) to this extension node.
    path []byte
    // The path (starting at pathBounds[0] and ending at pathBou
    // inner nodes that this single extension node replaces.
    pathBounds [2]byte
```

> 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.

5. `Question`: 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?

> 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

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

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

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

> 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.

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

```
// Set sets/updates the value for a given key
func (store *badgerKVStore) Set(key, value []byte) error {
    err := store.db.Update(func(tx *badgerv4.Txn) error {
        return tx.Set(key, value)
    })
    if err != nil {
        return errors.Join(ErrBadgerUnableToSetValue, err)
    }
    return nil
}
```

> 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.

9. `Question`: 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”?