Exposing Merkle Trees and Cryptographic Proofs

haruxe 0x60FF October 28th, 2022

Mint

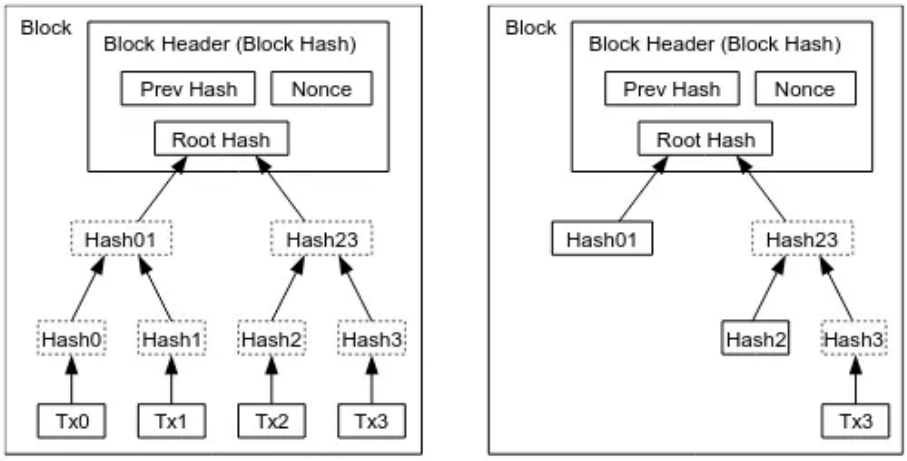
Introduction

In 1989, *Ralph Merkle* invented and patented the **Merkle Tree**, a data structure built for securing data with digital signatures. The structure was created for validating individual parts of a data set without needing the entire set of data to compare to; using less memory and much less computation time spent on larger sets of data.

Satoshi Nakamoto was the first to bring this technology to the mainstream in Cryptocurrency as the base structure for ensuring consistency of the Bitcoin network utilizes Merkle Trees. This is demonstrated in the Bitcoin Whitepaper which was published in 2008.

7. Reclaiming Disk Space

Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a Merkle Tree [7][2][5], with only the root included in the block's hash. Old blocks can then be compacted by stubbing off branches of the tree. The interior hashes do not need to be stored.



Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto, 2008, p. 4

Merkle Trees are composed of `Leaves`, `Branches`, and the `Root Hash`. For example, in a blockchain setting, a block's leaves may be composed of hashed transactions, which is then consequently hashed with neighboring hashes until it reaches the `Root`. The `Root Hash` is incredibly useful, it both verifies accuracy of previous and consequent blocks *and* can be used to validate the existence of a certain transaction in a block.

The magic of Merkle Tree's memory optimization lies in the fact that that a leaf can be verified by using only a number of proofs equivalent to `log2(n)` of a Tree of `n` size. For example, to prove validity of a leaf in a Merkle Tree of size `64`, you only need to provide 6 hash proofs in total (`log2(64) = 6`).

Smart Contracts

In the smart contract setting, **Merkle Trees** are most commonly used for **allow-lists** and providing a layer of **anonymity** (more detail later). One of the most common cryptography-related data structures is the Merkle Tree, so understanding how it works at least at a base level is incredibly essential for developers and security researchers alike. There are many different implementations of the Merkle Tree, but let's take a look at *OpenZeppelin's* MerkleProof.sol that is most commonly used.

```
function verify(
        bytes32[] memory proof,
        bytes32 root,
        bytes32 leaf
    ) internal pure returns (bool) {
```

```
        return processProof(proof, leaf) == root;
    }
```

The `verify()` function takes three arguments: `proof[]`, `root`, and `leaf`. Since we have the foundation knowledge of how the Merkle Tree works, we know that the `root` is the hash of the entire data set, the `proof[]` is all of the necessary “proofs” (aka additional hashes) to re-build the root with the `leaf` provided - which is the data that we wish to verify is in fact found in the data set.

Inside the `verify()` function, `processProof()` is called with the `proof[]` and `leaf`, which will in the end be compared to the root hash. Lets take a closer look on how it generates the `computedHash`.

```
function processProof(bytes32[] memory proof, bytes32 leaf) internal pure returns (b
        bytes32 computedHash = leaf;
        for (uint256 i = 0; i < proof.length; i++) {
            computedHash = _hashPair(computedHash, proof[i]);
        }
        return computedHash;
    }

function _hashPair(bytes32 a, bytes32 b) private pure returns (bytes32) {
        return a < b ? _efficientHash(a, b) : _efficientHash(b, a);
    }

    function _efficientHash(bytes32 a, bytes32 b) private pure returns (bytes32 valu
        assembly {
            mstore(0x00, a)
            mstore(0x20, b)
            value := keccak256(0x00, 0x40)
        }
    }
```

First, the `computedHash` is set to the `leaf` variable, which if you remember is the data that you wish to verify. This `computedHash` variable is declared for the purpose of providing a variable to be hashed over and over with itself until it reaches the `Root` hash.

On the next line, this `for` loop iterates over each `proof` that we provided. A `proof` is simply a hash that is required to rebuild the `Root Hash`, and is simply the neighboring node hash. For each `proof` provided in the array, the `computedHash` is set to be equal to `_hashPair(computedHash, proof[i])`, so `computedHash` changes each time and in the end will be built to be the `root`.

Following the function calls, we can see that the `_hashPair()` function checks `computedHash < proof[i]`. The reason this is checked has to do with how hashes are generated for each

subsequent branch of the tree. If `computedHash` is less than the current `proof`, it will be placed before the `proof` when hashed. The order that they are placed before hashing is necessary for the hashes to be performed correctly.

Inside the `_efficientHash()` function, you don't have to know how exactly the assembly works - but just know that the function simply returns the keccak256 hash of `(a, b)` in a more gas-efficient manner.

Example

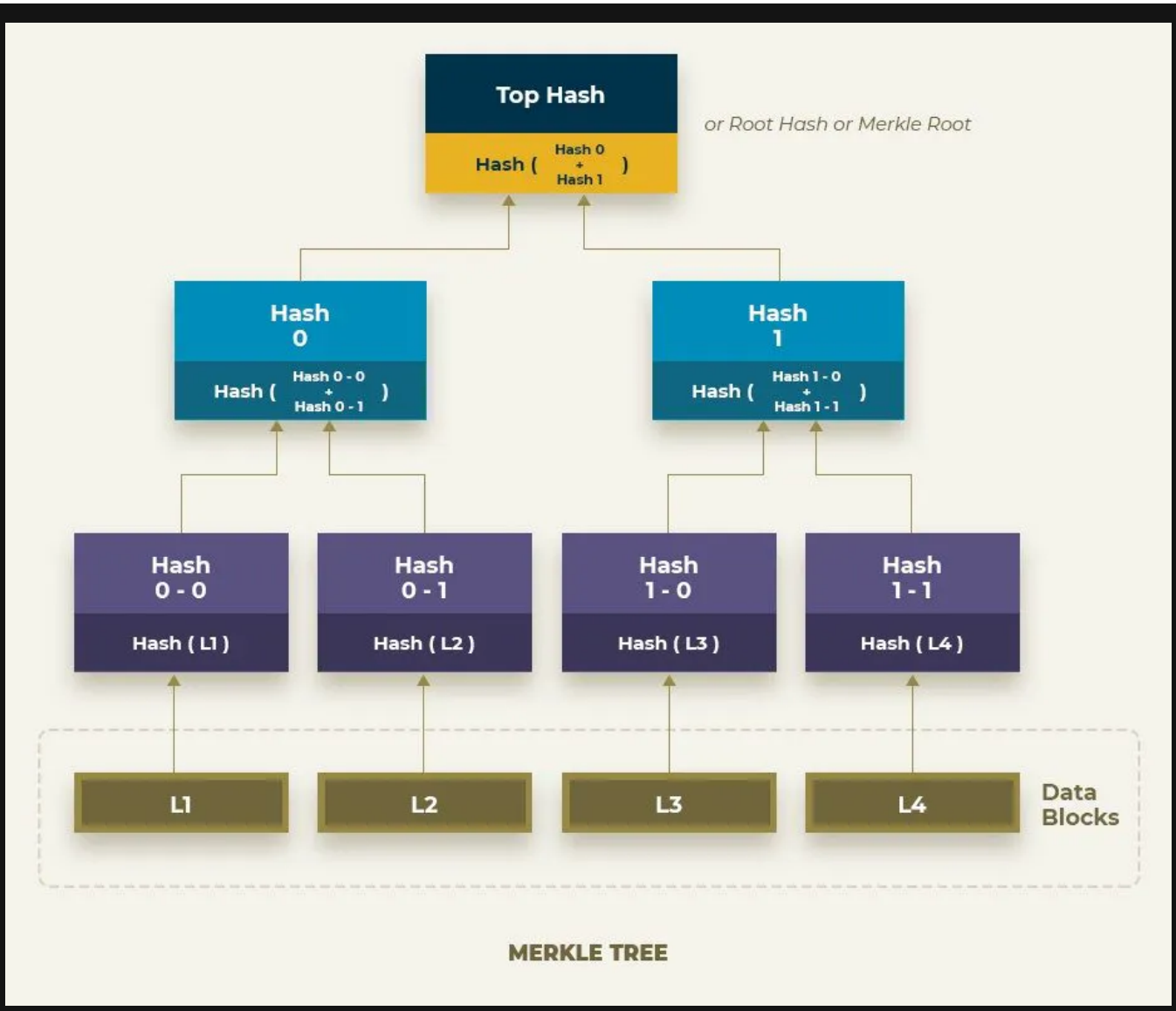
Lets say we had a whitelist of 4 members, composed of user addresses that we want to have exclusive access to our token - we hash those values in Merkle Tree fashion and get the following hash:

```
bytes32 private rootHash = 0x006b3e710f3089a74ecb6b0f5948e5ff07a3db6ba4da475d2be1762
```

When a user calls our `mint()` function, we call `MerkleProof.sol`'s `verify()` function, feeding it these values:

```
bytes32 userHash = keccak256(abi.encodePacked(msg.sender));
verify(proofs, rootHash, userHash);
```

The proofs are provided on the front-end before execution which is beyond the scope of this post - so we will focus on more of the internal structure.



Merkle Tree Diagram, Helios Solutions, 2020

First, lets say the `Leaf` we wish to verify is `Hash 0-1`. The first proof in the array provided to the function hashes `Hash 0-1` and `Hash 0-0` together which outputs `Hash 0`. This result is hashed with our second proof, `Hash 1`, which outputs the `Top Hash`, otherwise known as the `Merkle Root`.

The `Merkle Root` outputted from this calculation is compared with the root hash that was previously created, and if they are the same that means that `Leaf` does indeed exist in the data set!

Vulnerable Implementations

Merkle Trees are in themselves entirely secure. A `Root Hash` can be disclosed to anyone and everyone with no risks associated with modification of the data. This makes searching for vulnerable implementations a bit more difficult, so you will need to look at how each contract handles the Merkle Tree.

The most common vulnerability associated with Merkle Tree signatures is the `replay attack`. This is when a user is able to re-use a previously validated signature once more to exploit the contract.

A Merkle Tree in itself does not handle repeat uses of the same verification, so that is something that needs to be handled on a case-by-case basis.

The most common way to mitigate this is by using a `nonce`, which means a "number used once" to validate whether the signature has already been used. If you don't see any sort of `nonce` associated with verification, it may be a sign of a potential vulnerability.

Tornado Cash

Tornado Cash is one of the most popular Ether mixers on the Ethereum network. It utilizes Merkle Trees to validate ownership of each withdraw receipt without disclosing which leaf was used. How is this done?

ZK-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) is a technology that is used on top of the Merkle Tree data structure by Tornado Cash to ensure validity of the information entered into the contract without disclosing the actual information itself.

We won't go into the specifics of ZK-SNARK, but in essence it allows users to prove validity of a signature without sharing the signature itself. Tornado Cash uses *Circom* and *Snark.js* to generate the signatures. ZK-SNARK is already being used in other L2 chains and roll-ups and we will only see more of it because of it's incredible privacy properties.

The Tornado Cash's Merkle root is a hash that is ever-changing; it changes, breathes, and provides security for thousands of users.

Essentially, the Tornado Cash Merkle Tree smart contract generates a large amount of `Leaf` nodes that are `keccak256('tornado')`. When a user deposits Ether into the contract, the leaf is replaced with the hash of a `secret` string that is generated on the front-end, and a `nullifier` number that prevents a user from withdrawing multiple times.

Withdrawing from Tornado Cash is as simple as providing the `secret` and `nullifier` into the ZK-SNARK, which checks validity of your request. The Tornado Cash network does not know which funds you withdrew, blurring the lines between withdraw requests and deposits - keeping everyone anonymous.

Closing Thoughts

Having a thorough understanding of how Merkle Trees work and basic cryptographic knowledge will help you navigate foreign code quicker and with less confusion. For security researchers it is incredibly important to **verify** the code being reviewed rather than just **trusting** that the Merkle Tree implementation is correct and without bugs.

Merkle Trees serve a very clear purpose for basic signature validation - it uses less gas than the alternative allow-list when using a small-moderate size, and can hold any arbitrary data in addition to just a basic `address` or `uint`.

If you have any additional questions, send me a DM on twitter @haruxeETH and I will do my best to help you understand.

Keep building, keep hunting, and keep working hard to achieve your goals. - cheers.

Verification

This entry has been permanently stored onchain and signed by its creator.

ARWEAVE TRANSACTION ↗

hMGHhcrvpWrc0S-…WPHyjrohUm19_kU

AUTHOR ADDRESS ↗

0x60FF4545C6e674f…66143002Fa3A03C

CONTENT DIGEST

Gg7UG4hctOHyteV…oCs8uuiWx3WwVz4

More from haruxe