Cryptographic Tools 101 - Hash Functions and Merkle Trees Explained

What's This Article About?

Blockchains enable humans to agree on things without requiring intermediaries. Instead of relying on trust, blockchains rely on cryptographic proofs. Cryptographic primitives are used to provide these proofs. But what are they?

In this article we look at two cryptographic primitives that are essential to cryptographic proofs on blockchains: hash functions and Merkle trees. We'll explore the core mechanics of hash functions, see how they're important to blockchains, and learn about hash pointers. Then we'll examine traditional and concurrent Merkle trees, outlining their importance to Solana.

What's a Cryptographic Primitive?

A cryptographic primitive is an operation or algorithm that is fundamental to building cryptographic protocols and systems. Cryptographic primitives to cryptographic protocols are like atoms to molecules - they are the building blocks for more complex solutions. Random number generators, commitment schemes, and public key cryptography are all examples of cryptographic primitives.

Alone, cryptographic primitives are quite limited. Combined, cryptographic primitives provide basic security functions such as authentication, confidentiality, and integrity. Combining cryptographic primitives is a very nuanced process that requires lots of careful planning and a deep understanding of how each primitive interacts with one another. In this process, you must be mindful of security considerations in light of the security goals you wish to achieve. The methods for combining cryptographic primitives can be categorized broadly as:

- Sequential composition

: Applying one primitive after another (e.g., hash chaining)

- Parallel Composition

: Using primitives simultaneously and independently (e.g., encrypting and hashing data simultaneously)

- Hierarchical Composition

: Using a cryptographic primitive inside of another (e.g., Merkle trees)

Understanding what a cryptographic primitive is, how they work, and the subtleties involved when combing them helps you understand and architect systems that are secure and efficient. One of the most commonly used and combined cryptographic primitives is a hash function.

What's a Hash Function?

A hash function is a cryptographic function that takes data of any size and returns a fixed-size value. The value returned by this function is referred to as a digest, or a hash

. Popular hashing algorithms include: SHA-1, SHA-2, SHA-3, MD5, and Argon2. Hash functions are used everywhere throughout blockchains, so it is vital that you understand what they are and how they work.

A Simple Analogy

Imagine baking an extravagant chocolate cake. The cake has multiple layers, each made with their own ingredients. While baking, your friend texts you and asks what you're up to. Giving your friend a detailed explanation about how you made the cake, and all the ingredients used would be quite cumbersome. Instead, you decide to send your friend a picture of the chocolate cake.

Here, the picture of the cake serves as a hash - it's a simple, compact representation of something much more complex. Your friend wont know every last ingredient used to make the cake, but they'll have a good idea of what you've just baked. Say your chocolate cake had raspberries on top of it. If you took them off or switched them out with strawberries then the picture of your cake would be completely different from the final product. Similarly, any changes to the data hashed would produce a new hash value.

Properties of a Good Cryptographic Hash Function

Admittedly, the definition of a hash function given previously is misleading. A hash function could return a variable-sized

hash. It could also return the same hash for two different inputs. It could also be extremely easy for someone to reverse-engineer the hash and find the original input. The definition given earlier was for a good

cryptographic hash function. But what makes a cryptographic hash function a good

cryptographic hash function?

A good hash function is deterministic - the same input will always create the same output. If I hash the input “baseball”, then, regardless of the system, that hash function will output the same hash every time. This also means, partly, that the hash will be the same size regardless of the input’s size. This is important for processing efficiency and data storage. Knowing a unique input will always create a unique output, and these outputs will always be of fixed size, is a clear indicator of a good cryptographic hash function.

A good hash function is pre-image resistant. This means that it is infeasible computationally to reverse-engineer the input value based on its hash. So, if a person gives you a hash then you shouldn’t be able to figure out what data produces that hash. This also leads into the idea that two distinct sets of data should not produce the same hash. A good hash function is said to be collision resistant when any two inputs never

produce the same hash.

A good hash function subscribes to the Avalanche Effect - a small change to the input should result in a drastically different hash. Changing even a single character should produce a completely different hash. Thus, a hash output should not reveal any information about the input or have any recognizable patterns. Notice the difference in hashes in the figure above. The red fox “runs” creates a radically different hash compared to the red fox “walks”. There’s also no indication that these two hashes contain almost identical information.

A good hash function should be fast to compute. Slow hash functions are not practical for real-time or near-real-time computation situations such as transaction verification. A slow hash function could become a serious bottleneck, limiting both throughput and network performance. A fast hash function is necessary for efficient and secure performance on the blockchain.

Why is this Important to Blockchains?

A blockchain is a decentralized, distributed ledger that records transactions across its network. These transactions are grouped together in blocks, which are securely linked via a good

hash function. Each block contains transaction data, a timestamp, and the hash of the previous block. Because each block’s hash is dependent on the previous block’s hash, any change made to a block’s contents would change its hash and invalidate all subsequent blocks. This process of using previous hashes in the generation of a new hash is known as hash chaining.

Adding a new block to a blockchain is known as a confirmation. A confirmation verifies and secures all the transactions in the new block as well as all previous blocks. This is because for every new confirmation the harder it is to alter previous blocks. To alter a previous block, an attacker would have to recompute all previous hashes. Thus, hash chaining ensures it would be infeasible to alter the blockchain once a block has a considerable amount of confirmations.

Simply put, a blockchain is a chain of blocks secured via hash functions. But how exactly do we point from one block to another? Yes, a cryptographic hash function is used to chain the blocks together but how are we able to look at the data from previous blocks? I thought a good

cryptographic hash function is pre-image resistant.

What’s a Hash Pointer?

A pointer is a variable that holds the location to where specific data is stored in memory. The data at this memory address can be accessed easily since the pointer “points to” the location of the data. A hash pointer is a data structure similar to a pointer, but it also contains a cryptographic hash of the data referenced. Thus, a hash pointer tells you where to access a specific piece of data and allows you to check the integrity of the data accessed.

The structure of a blockchain can be more accurately described as a linked list that uses hash pointers. The hash of the previous block is a hash pointer that points to a set of transactions and a hash of all those transactions. Hash pointers facilitate block linking, the integrity of each block, and verifying that newly added blocks correctly follow the previous blocks.

Hash pointers are used to efficiently chain blocks together. But what about the transactions in the block? If a block contained a thousand transactions, wouldn’t it be expensive to verify these transactions individually?

What’s a Merkle Tree?

A Merkle tree is a data structure used to organize and verify large sets of data. The data is organized into a tree-like

structure where each leaf, or node, is labelled with the hash of a set of data. Each non-leaf node is a hash of its child nodes. Merkle trees are used to verify transactions included in specific blocks that are propagated to the blockchain. So, how does it work?

Transactions are batched together in a list to form a block. Each transaction in the list is hashed using a good

hashing function. These hashes serve as the leaf nodes. These leaf nodes are hashed together in pairs to create a new layer of hashes. This process continues iteratively until there is a single hash remaining known as the Merkle root. The Merkle root is stored in the block's header and acts as a digital fingerprint of all the transactions within that block. We can also refer to the Merkle root as the block's hash. So when we say a new block is linked with a previous block using the previous block's blockhash, we are saying the block uses the previous block's Merkle root as a part of the new block's hash.

Merkle trees make verifying individual transactions within a block efficient. Traditionally, you'd have to verify each transaction to verify a specific transaction, which is expensive and time consuming. Merkle trees provide a cryptographic "shortcut" to aid in this verification process known as a Merkle proof. A Merkle proof is a path from the transaction's leaf node all the way up to the Merkle root. Using the figure above, think of it as a path from Data A to the Merkle root. This path also includes its sibling nodes, the leaves adjacent to each node on the path but are not part of the path themselves. A verifier can compute a hash using the proof path to check whether their computed hash matches the Merkle root. If the resulting hash matches, the verifier is confident that the transaction is legitimate and has not been tampered with.

A leaf can be changed by hashing the new leaf's data and recomputing the Merkle root. This new Merkle root is used to verify the new changes and invalidates the previous proof. In a high throughput network such as Solana, validators can receive change requests to on-chain Merkle trees in rapid succession (e.g., within the same slot). Each data change would need to be recomputed sequentially or else each subsequent change request would be invalidated by the previous change request made within the slot. Changing leaf data and computing a new Merkle root is very common in blockchains. So how do we address rapid changes?

What's a Concurrent Merkle Tree?

A concurrent Merkle tree is a Merkle tree that is optimized for concurrent reads and writes. It stores a secure changelog of the most recent changes, their root hash, and the proof to derive it. This changelog is stored on-chain in an account specific to the tree with a maximum number of changes that can occur to it while the Merkle root is still valid. This maximum number of changes is known as the maxBufferSize

. Thus, when a validator receives change requests to an on-chain Merkle tree in rapid succession, it can use this changelog as a source of truth allowing for up to maxBufferSize

changes to the tree in the same slot.

Concurrent Merkle trees improve upon traditional Merkle trees making them suitable for high throughput environments such as Solana. To create an on-chain concurrent Merkle tree on Solana, there are three properties that affect the size of your tree, the cost to create your tree, and the number of concurrent changes to your tree:

- Max depth
- Max buffer size
- Canopy depth

Max depth refers to the maximum number of hops it takes to get from any leaf to the Merkle root. maxDepth is used to determine the maximum number of nodes to store within the tree. This can be calculated using the formula: numberOfNodes = 2 ^ maxDepth

. A tree's depth must be set at creation, so it is important to use this formula to determine how many pieces of data you'd like the tree to store.

As previously stated, max buffer size refers to the maximum number of changes that can be made to a tree with its Merkle root still being valid.

Canopy depth refers to a subset of the Merkle tree that is stored on-chain. These cached proofs are used to check a hash against the on-chain Merkle root. The complete proof path must be used to verify original ownership of a leaf when performing a write to it. For example, you would want to write to the tree if you were transferring an NFT. The canopy allows you to reduce the proof size and avoids you from using a proof size of maxDepth

to verify the tree. A tree with a maxDepth

of 20 would require a proof size of 20. With a canopy of 15, only a proof size of 5 is required to be submitted per write transaction. Thus, a higher canopy depth results in paying a higher upfront cost so you can submit smaller proofs later on.

Canopy depth is a leading factor in the cost of creating a tree. This is because the higher the canopy depth, the larger the account required. Developers can use the @solana/spl-account-compression

package to calculate the required space for a given tree size and the cost to allocate the required space for the tree on-chain. Developers can use the getConcurrentMerkleTreeAccountSize

function to calculate the required space for a given account based on its parameters, and use getMinimumBalanceForRentExemption

on the required space to get the final cost in Lamports.

Solana utilizes concurrent Merkle trees for state compression. State compression is the method of creating a hash of off-chain data and storing it on-chain for secure verification. The most popular use case for state compression is compressed NFTs because it dramatically reduces the cost of minting. For example, minting a billion NFTs on Solana would cost 507 $SOL compared to 12 000 000 $SOL with “regular” NFTs. Now that you have a solid understanding of hashing and Merkle trees, we’ll delve into compressed NFTs in a future article!

Conclusion

Congratulations! In this article we analyzed hashing functions and Merkle trees, two cryptographic primitives that are vital to blockchains. Understanding blockchains is no small feat - they are complex distributed systems that require a breadth of technical understanding. Oftentimes, this knowledge is assumed for the average developer, user, or investor. This article does not assume any prior knowledge of cryptographic primitives. Instead, we begin with the basics before progressing into more complex discussions on traditional and concurrent Merkle trees. It is crucial to have this fundamental understanding as we prepare to delve into more complex topics such as compressed NFTs. With this new knowledge, you are better equipped to navigate codebases or discussions regarding cryptographic primitives and more complex cryptographic solutions.

If you’ve read this far anon, thank you!

Additional Resources / Further Reading

- Detailed book chapter on cryptographic primitives
- Article comparing secure hashing algorithms
- Hashing Algorithms and Security - YouTube video by Computerphile
- Concise YouTube video on Merkle trees
- Exploring NFT Compression on Solana