# Lecture 2: Blockchains as Cryptographic Data Structures

Principles of Blockchains, University of Illinois,
Professor: Pramod Viswanath
Scribe: Suryanarayana Sankagiri

January 28, 2021

**Abstract**

There are two main cryptographic tools used critically in blockchains: **cryptographic hash functions** and **digital signatures**. We first briefly review regular hash functions, and their use in hash tables. We then introduce cryptographic hash functions and state their additional security properties. The notion of a **hash pointer** is then introduced, which is used to define the data structures **blockchains** and **Merkle trees**. We cover properties of Merkle trees as relevant to blockchains in detail leading to the creation of a tamper-resistant ledger. Finally, we present digital signatures (the public-key variant) used to authenticate messages (i.e., map a message to its sender) setting the stage for a decentralized ledger.

## Hash Functions

A hash function is a function that converts a binary string of arbitrary length to a binary string of fixed length. Since any data structure can be represented as a binary string, a hash function converts any data structure to a fixed-length binary string. For any piece of data $x$, the output of the hash function is denoted by $H(x)$ and is called the hash of $x$. Given a hash function, suppose its output is always a binary string of length $k$. Then the size of the output space is $2^k$. Call this output space $Y$.

A good hash function has two properties. Firstly, it should be very fast to compute. Secondly, it must reduce the chance of 'collisions' over the expected input space. A collision happens when there are two inputs, $x$ and $x'$, which get mapped to the same hash, i.e., $H(x) = H(x')$. To minimize collisions, a hash function must distribute its output uniformly over the output space $Y$. Let us elaborate on this. Suppose we restrict ourselves to some fixed set of inputs (denoted by $X$). Then, if we choose an input $x$ uniformly at random from this set, the probability that $H(x)$ is equal to a particular $y$ should be close to $2^{-k}$. Given such a hash function, and $m$ inputs that are sampled uniformly at random from $X$, what is the probability that there will be a collision? The 'birthday paradox' comes into play here. The probability that there will be at least one collision among the $m$ hashes is approximately $1 - \exp(-m^2 2^{-k})$. Thus, for $m \approx 2^{k/2}$, there is a reasonable chance for a hash collision.

A classic use of hash functions is to create **hash tables**. Hash tables are a form of a 'key-value store', just like 'dictionaries' in standard programming languages. They are used to reduce the lookup time for values stored in it. In what follows, a value is a piece of data; it could be in the form of any data type (integer, string, tuple, or a custom type). The following example illustrates why hash tables are useful. Suppose we want to store $C$ (say, a million) values, e.g., (name, phone number, email address) tuples. We could store them as a list of length $C$, which would take $O(C)$ time to look up. Instead, we could first hash the values into a binary string of length $k$, and store each value that hashes to some particular $y \in Y$ as a separate list. These lists would be of size $C/2^k$,

because of the uniform spread of the hashes. These shorter lists would take much lesser time to look up. The additional overhead of computing the hash is minimal.

The Wikipedia page on hash functions describes the use of hash functions in hash tables in more detail and gives many examples of hash functions. For example, one can take the last 256 bits of a long bit string as its hash. More generally, one can take any subset of 256 (or $k$) bits as the hash. A potential drawback of such hashes is that there could be many collisions; this depends upon the input set. One can choose the hash to represent the subset of bits in a bitstring that are likely to change from one data point to another in the relevant application. One can also perform certain computations that mix-up the bits, such as computing the xor of the first and last bits, the second and second-last bits, and so on.

## Cryptographic Hash Functions

In cryptocurrencies, we require additional properties from a hash function, which are captured via a **cryptographic hash function**. The key feature of a cryptographic hash function is that it is easy to compute, but difficult to invert. To elaborate, given a binary string $x$, it is easy to compute $y = H(x)$, but given an arbitrary $y'$, it is difficult to find any $x'$ for which $H(x') = y'$. By difficult, we mean that it would take a really long time to find such an $x'$. Essentially, the only way to do so is to exhaustively search over all possible values of $x'$. We say that such hash functions have **collision resistance**. This is because the above property implies that it is difficult to come up with two values $x, x'$ that are not equal such that $H(x) = H(x')$. The output space of cryptographic hash functions must be large, or else one could easily find a hash collision by iterating over the output space. Typically, they are 128-bit strings, 256-bit strings, or longer. A hash function's security is determined by how easy it is to find hash collisions (or partial hash collisions, defined in terms of equality of a prefix of the hash).

Note that regular hash functions minimize the chance of collisions if the inputs are chosen at random. However, if one is specifically looking to find a hash collision, it is easy to do so (see if you can find collisions for the examples given above). In contrast, for a cryptographic hash function, finding collisions should be practically impossible, even for an adversary specifically trying to do so. In blockchains we will exclusively use cryptographic hash functions and we refer to cryptographic hashes as merely hashes. It is useful to think of a hash function as a random oracle, which generates an arbitrary random $k$-bit string for any input with a uniform distribution. Note that the oracle always returns a fixed output for a given input. In practice, for $k$ large enough, we assume that collisions never occur.

Constructing a cryptographic hash function is not easy (in contrast to regular hash functions). In fact, they should not be created in an ad-hoc manner. The National Institute of Standards and Technology (NIST) sets a standard for these hash functions. One such function is **SHA-256**, where SHA stands for Secure Hash Algorithm, and 256 is the length of the output. This function has been studied extensively and is currently believed to be secure enough for practical applications. The exact construction of this function, and the best-known attacks on it, are given on the Wikipedia page for SHA. This function is available as part of a 'cryptography' library in most standard programming languages.

A general principle used in the construction of many cryptographic hash functions is the **Merkle-Damgard construction**. A rough description of this construction is as follows. Suppose we are given a function $F$ that compresses strings of a certain length $l$ (say 512) bits to strings of shorter length $k$ (say 256) bits. Further, suppose this function has adversarial collision resistance (i.e., it is a cryptographic hash function). Then this function can be used to create a cryptographic hash function $H$, which takes as input an arbitrarily long string $m$. Break $m$ into portions of length $l - k$ (256 in our example); call these portions $m_1, m_2, \ldots$. We appropriately pad the message with some extra bits to ensure that all portions are of equal length. The compression function $F$ is applied

recursively, with the output of the $i^{\text{th}}$ iteration padded to $m_{i+1}$ and fed into the $i+1^{\text{th}}$ iteration of $F$. The initial input is a fixed string of $l$ bits.

## Uses of cryptographic hashes

A hash of a certain value acts as a **commitment** for that value. Consider an auction, where each party would like to bid a certain price without openly revealing the value to all. One can broadcast the hash of the value instead of the value itself. Later, after all the bids have been placed, one can reveal the actual value and others can verify if it was the originally committed value. Due to the collision resistance property, one cannot generate an alternate value that matches the same hash value; one is committed to the original value. Thus, publishing the hash of a value is like writing it on a piece of paper and placing it in a sealed envelope.

A hash function can also be used as a *pointer* to certain values when these are stored in a hash table. Due to the collision resistance property, each value in the hash table has a unique hash. Thus, the hash serves as a pointer to the value within the hash table. When used in this fashion, we call the hash a **hash pointer**. Note that a hash pointer is nothing more than a hash; the term alludes to the fact that the hash is being used as a pointer.

A third application of hashes is that we can use it to create **hash puzzles**. The problem (or puzzle) is to find an input $x$ such that $H(x)$ is less than a certain threshold. The simplest form of the threshold is to require that the first $n$ bits of the hash are all zero. The best method of solving such a puzzle is simply to randomly choose different inputs. This variable input to a hash puzzle is called a **nonce**. More generally, the input is a tuple (nonce, data), where data stands for any useful data. For any threshold $\tau$, the probability that a certain nonce succeeds is $\tau/|Y|$ (recall $Y$ is the output space of the hash). Thus, on average, one needs to try $|Y|/\tau$ different nonces to succeed. If $|Y|/\tau$ is set to be a very large number, say $10^10$, and a computer can compute 1 hash in 10 nanoseconds, then such a computer would be able to solve the hash puzzle in 100 seconds (on average). Note that it is very quick to verify that a certain (nonce, data) tuple solves the hash puzzle. Hash puzzles have been used as a means of preventing spam. If a sender must solve a hash puzzle each time it sends an email, then it cannot do so very frequently. Here, the data field could be the email and recipient's address, so it needs to recompute the puzzle every time it sends a new email.

## Blockchains

The blockchain data structure is a linked list that uses hash pointers instead of regular pointers. In the context of blockchains, a block is a data type that contains a particular header field, called the 'hash pointer,' and some 'data'. The hash pointer field is simply the hash of another block, which we call its parent. A sequence of such blocks forms a chain, with each block containing a hash pointer to its parent; we call this chain a blockchain. A blockchain system is much more complicated than the blockchain data structure. Since this data structure lies at the heart of digital trust systems, the same term is used for the whole system. (Figure forthcoming).

Blockchains are tamper-proof data structures, making them particularly useful in digital trust systems. Generally speaking, each party in the system stores a local copy of this data structure in the form of a hash table. Using the hash pointers, a party can obtain the sequence of ancestors (parent, parent's parent, and so on) of any given block. Suppose a particular block is missing from a party's local hash table. It can query its peers for the block using the block's hash (obtained from its child). It can then verify that the block it receives from its peer is the correct one (i.e., it has not been tampered with) by checking that its hash matches with what it has; this is essentially using the hash as a commitment. Extending this principle, a party can check if any portion of the blockchain it receives has been tampered with or not.

Sometimes it is useful for a party to verify the membership of one particular data value in the blockchain. Having access to the full blockchain and all data internal to it, will naturally provide proof of such membership. But for a party that is only interested in verifying the membership of one particular data value, this is very onerous. In anticipation of this requirement in practical blockchain systems, we consider a Merkle tree data structure to organize the data inside each of the blocks. This is discussed next.

## Merkle Trees

A Merkle tree is a directed tree formed using hash pointers. It is constructed from a set of data values as follows. The hash of each value forms the *leaf node* of the Merkle tree. An *internal (non-leaf) node* contains the hashes of its two (or more) children nodes. In other words, a parent node consists of the hash pointers to its children. These tree nodes converge to a single root node. See the Wikipedia page on Merkle trees for a more elaborate explanation with a figure.

Merkle trees are also tamper-proof data structures. By storing only the hash of the root of the Merkle tree (root hash), one can detect any modifications to the tree. However, they have additional properties, namely that of an accumulator (in some places, the term authenticated data structure is used instead of accumulator). We will not formally define these terms but rather provide a simple explanation. An accumulator scheme provides a compact commitment for a set of values. In addition, it provides a means to prove whether a specific value is part of the committed set without revealing the other values. For Merkle trees, this compact commitment is the root hash. The tree's branch from the root node to a leaf node acts as a proof/witness for the corresponding value being in the set. See here for a more detailed explanation of Merkle trees as an accumulator, with figures.

## Blockchains as a Ledger

A **ledger** is an ordered list of data values. The blockchain data structure provides a tamper-resistant ordered sequence of blocks. Storing the data in each block as a Merkle tree allows for a natural ordering (e.g., lexicographic) of the data values. Put together, the two data structures (blockchain and Merkle tree) lead to a tamper resistant ledger. We did not specify who is creating the ledger so far; presumably a single party has "write" permission with all others having "read" permissions. This describes a centralized ledger above, with one person writing and many persons reading. How do we enable many parties to write? We first need a way to distinguish different parties. Digital signatures are a method to provide people an identity, using which they can write blocks to the blockchain. This basic cryptographic primitive is discussed next.

## Digital Signatures

Digital signatures are the cryptographic analog of handwritten signatures. Broadly speaking, a signed message allows anyone to check who the message's sender is. To elaborate, a digital signature scheme gives a pair of keys to each user: a secret key, which is held privately by each user, and a public key, which is given to everybody. A message can be signed using one's secret key, and the signature is sent along with the message. A different user can verify that the message indeed came from its purported sender by matching the message and signature to its public key. Thus, a user's public key becomes the user's identity in the system. The scheme is secure if an adversary cannot forge signatures, i.e., it cannot generate a valid signature without knowing the corresponding party's secret key.

An important point to note is that the signature is different for each message. Otherwise, upon receiving one particular message from a user, an adversary can then simply append the signature

to other messages, rendering the scheme pointless. Typically, users sign the hash of the message that they wish to send. Thus, the object being signed is a constant-length bit string, and so is the signature itself. Once again, for unforgeability, one must have that the signature is long enough.

Just as in hash functions, there are standards for secure signature schemes, declared by NIST. The scheme used by Bitcoin is known as the Elliptic Curve Digital Signature Algorithm (ECDSA). Just as in the case of hashes, these signature schemes have been empirically tested for years, and there is good reason to believe they are secure. We will not specify what elliptic curves are, and how they are used for signatures here. You can learn more about them from Wikipedia page on ECDSA and associated links, as well as from the references given below.

The first application of digital signatures is to upgrade the centralized blockchain system we saw earlier to a decentralized version. We do this by adding a digital signature to each block; this allows different users to append blocks to the blockchain and identify themselves through the signature. *Figure forthcoming.* There are three questions in this decentralized architecture:

1. Who are the set of users that can participate in and how are they chosen?
2. When and which block does a user get to append and how do others verify this rule in a decentralized manner?
3. Where does a user append the block? In principle, a block can be appended to any other block in the view of the user.

Blockchain designs and their properties vary in how they answer the questions to these three questions. In the next lecture we will see how Bitcoin resolves these three questions.

The second application of digital signatures is in providing user attribution to the data stored in the blocks. So far the data is considered to be an abstract digital entity, but in many applications it makes sense to have attribution to users. For instance, in cryptocurrencies, the data refers to transactions which record change of ownership of coins. Now using signatures for users ownership of coins can be established (during creation stage) and during transfer of ownership). *Figure forthcoming.*

## Summary

A hash maps any value (piece of data) to a constant-sized bit string. For practical purposes, the hash of a value uniquely identifies it. Hashes are used to construct blockchains and Merkle trees, which are tamper-proof data structures. Hashes provide commitments to pieces of data. In addition, Merkle trees act as accumulators, enabling one to provide a proof of membership for individual elements in a committed set. Digital signatures parallel hand-written signatures and thereby provide authenticated communication. All messages that are exchanged in a blockchain system are signed.

We also described a ledger, which is an ordered list of data values. We saw how the blockchain (and Merkle tree) data structure are sufficient to create a centralized ledger, with a central authority writing to the ledger and multiple parties reading from it. We discussed how digital signatures help us move towards a decentralized ledger. Finally, we identified three important questions to answer in order to have a functional decentralized ledger.

## References

A good reference for the material of this lecture is Chapter 1 of the book "Bitcoin and Cryptocurrency Technologies" by Narayanan et al. In particular, it elaborates on the use of Merkle trees as an authenticated data structure, and explains digital signature schemes in more detail. A free, pre-publication version is available at this link.

This public talk by Dan Boneh describes elliptic curves and their usage in cryptography, among other things.

The original patent application by R. Merkle is a superb place to learn about Merkle trees. It is not every patent application that is this lucid and clear.