Bitcoin and Cryptocurrency Technologies

Srinivas R Rao

Feedback welcome! Email raosrinivasr70@gmail.com

Introduction to the book
There's a lot of excitement about Bitcoin and cryptocurrencies. We hear about startups, investments, meetups, and even buying pizza with Bitcoin. Optimists claim that Bitcoin will fundamentally alter payments, economics, and even politics around the world. Pessimists claim Bitcoin is inherently broken and will suffer an inevitable and spectacular collapse.

Underlying these differing views is significant confusion about what Bitcoin is and how it works. We wrote this book to help cut through the hype and get to the core of what makes Bitcoin unique.

To really understand what is special about Bitcoin, we need to understand how it works at a technical level. Bitcoin truly is a new technology and we can only get so far by explaining it through simple analogies to past technologies.

We'll assume that you have a basic understanding of computer science — how computers work, data structures and algorithms, and some programming experience. If you're an undergraduate or graduate student of computer science, a software developer, an entrepreneur, or a technology hobbyist, this textbook is for you.

In this series of eleven chapters, we'll address the important questions about Bitcoin. How does Bitcoin work? What makes it different? How secure are your bitcoins? How anonymous are Bitcoin users? What applications can we build using Bitcoin as a platform? Can cryptocurrencies be regulated? If we were designing a new cryptocurrency today, what would we change? What might the future hold?

Each chapter has a series of homework questions to help you understand these questions at a deeper level. We highly recommend you work through them. In addition, there is a series of five programming assignments in which you'll implement various components of Bitcoin in simplified models. If you're an auditory learner, most of the material of this book is also available as a series of video lectures. You should also supplement your learning with information you can find online including the Bitcoin wiki, forums, and research papers, and by interacting with your peers and the Bitcoin community.

Chapter 1: Introduction to Cryptography & Cryptocurrencies

All currencies need some way to control supply and enforce various security properties to prevent cheating. In fiat currencies, organizations like central banks control the money supply and add anti-counterfeiting features to physical currency. These security features raise the bar for an attacker, but they don't make money impossible to counterfeit. Ultimately, law enforcement is necessary for stopping people from breaking the rules of the system.

Cryptocurrencies too must have security measures that prevent people from tampering with the state of the system, and from equivocating, that is, making mutually inconsistent statements to different people. If Alice convinces Bob that she paid him a digital coin, for example, she should not be able to convince Carol that she paid her that same coin. But unlike fiat currencies, the security rules of cryptocurrencies need to be enforced purely technologically and without relying on a central authority.

As the word suggests, cryptocurrencies make heavy use of cryptography. Cryptography provides a mechanism for securely encoding the rules of a cryptocurrency system in the system itself. We can use it to prevent tampering and equivocation, as well as to encode the rules for creation of new units of the currency into a mathematical protocol. Before we can properly understand cryptocurrencies then, we'll need to delve into the cryptographic foundations that they rely upon.

Cryptography is a deep academic research field utilizing many advanced mathematical techniques that are notoriously subtle and complicated to understand. Fortunately, Bitcoin only relies on a handful of relatively simple and well-known cryptographic constructions. In this chapter, we'll specifically study cryptographic hashes and digital signatures, two primitives that prove to be very useful for building cryptocurrencies. Future chapters will introduce more complicated cryptographic schemes, such as zero-knowledge proofs, that are used in proposed extensions and modifications to Bitcoin.

Once we've learnt the necessary cryptographic primitives, we'll discuss some of the ways in which those are used to build cryptocurrencies. We'll complete this chapter with some examples of simple cryptocurrencies that illustrate some of the design challenges that we need to deal with.

1.1 Cryptographic Hash Functions

The first cryptographic primitive that we'll need to understand is a ***cryptographic hash function*.** A ***hash function*** is a mathematical function with the following three properties:

- Its input can be any string of any size.
- It produces a fixed size output. For the purpose of making the discussion in this chapter concrete, we will assume a 256-bit output size. However, our discussion holds true for any output size as long as it is sufficiently large.
- It is efficiently computable. Intuitively this means that for a given input string, you can figure

out what the output of the hash function is in a reasonable amount of time. More technically, computing the hash of an n-bit string should have a running time that is $O(n)$.

Those properties define a general hash function, one that could be used to build a data structure such as a hash table. We're going to focus exclusively on *cryptographic* hash functions. For a hash function to be cryptographically secure, we're going to require that it has the following three additional properties: (1) collision-resistance, (2) hiding, (3) puzzle-friendliness.

We'll look more closely at each of these properties to gain an understanding of why it's useful to have a function that behaves that way. The reader who has studied cryptography should be aware that the treatment of hash functions in this book is a bit different from a standard cryptography textbook. The puzzle-friendliness property, in particular, is not a general requirement for cryptographic hash functions, but one that will be useful for cryptocurrencies specifically.

Property 1: Collision-resistance. The first property that we need from a cryptographic hash function is that it's collision-resistant. A collision occurs when two distinct inputs produce the same output. A hash function $H(.)$ is collision-resistant if nobody can find a collision. Formally:

Collision-resistance: A hash function H is said to be collision resistant if it is infeasible to find two values, x and y, such that $x \neq y$, yet $H(x)=H(y)$.

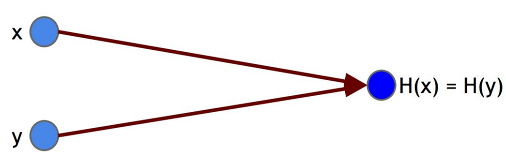

Figure 1.1 A hash collision. x and y are distinct values, yet when input into hash function H, they produce the same output.

Notice that we said *nobody can find* a collision, but we did not say that no collisions exist. Actually, we know for a fact that collisions do exist, and we can prove this by a simple counting argument. The input space to the hash function contains all strings of all lengths, yet the output space contains only strings of a specific fixed length. Because the input space is larger than the output space (indeed, the input space is infinite, while the output space is finite), there must be input strings that map to the same output string. In fact, by the Pigeonhole Principle there will necessarily be a very large number of possible inputs that map to any particular output.

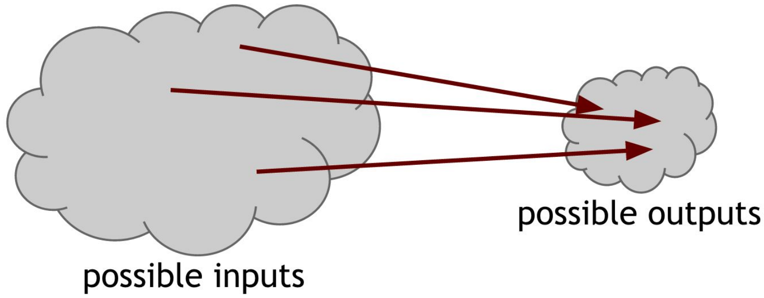

Figure 1.2 Because the number of inputs exceeds the number of outputs, we are guaranteed that there must be at least one output to which the hash function maps more than one input.

Now, to make things even worse, we said that it has to be impossible to find a collision. Yet, there are methods that are guaranteed to find a collision. Consider the following simple method for finding a collision for a hash function with a 256-bit output size: pick $2^{256} + 1$ distinct values, compute the hashes of each of them, and check if there are any two outputs are equal. Since we picked more inputs than possible outputs, some pair of them must collide when you apply the hash function.

The method above is guaranteed to find a collision. But if we pick random inputs and compute the hash values, we'll find a collision with high probability long before examining $2^{256} + 1$ inputs. In fact, if we randomly choose just $2^{130} + 1$ inputs, it turns out there's a 99.8% chance that at least two of them are going to collide. The fact that we can find a collision by only examining roughly the square root of the number of possible outputs results from a phenomenon in probability known as the ***birthday paradox***. In the homework questions at the end of this chapter, we will examine this in more detail.

This collision-detection algorithm works for every hash function. But, of course, the problem with it is that this takes a very, very long time to do. For a hash function with a 256-bit output, you would have to compute the hash function $2^{256} + 1$ times in the worst case, and about 2^{128} times on average. That's of course an astronomically large number — if a computer calculates 10,000 hashes per second, it would take more than one octillion (10^{27}) years to calculate 2^{128} hashes! For another way of thinking about this, we can say that, if every computer ever made by humanity was computing since the beginning of the entire universe, up to now, the odds that they would have found a collision is still infinitesimally small. So small that it's way less than the odds that the Earth will be destroyed by a giant meteor in the next two seconds.

We have thus seen a general but impractical algorithm to find a collision for *any* hash function. A more difficult question is: is there some other method that could be used on a particular hash function in order to find a collision? In other words, although the generic collision detection algorithm is not feasible to use, there still may be some other algorithm that can efficiently find a collision for a specific hash function.

Consider, for example, the following hash function:

$$H(x) = x \bmod 2^{256}$$

This function meets our requirements of a hash function as it accepts inputs of any length, returns a fixed sized output (256 bits), and is efficiently computable. But this function also has an efficient method for finding a collision. Notice that this function just returns the last 256 bits of the input. One collision then would be the values 3 and $3 + 2^{256}$. This simple example illustrates that even though our generic collision detection method is not usable in practice, there are at least some hash functions for which an efficient collision detection method does exist.

Yet for other hash functions, we don't know if such methods exist. We suspect that they are collision resistant. However, there are no hash functions *proven* to be collision-resistant. The cryptographic hash functions that we rely on in practice are just functions for which people have tried really, really hard to find collisions and haven't yet succeeded. In some cases, such as the old MD5 hash function, collisions were eventually found after years of work, leading the function to be deprecated and phased out of practical use. And so we choose to believe that those are collision resistant.

Application: Message digests Now that we know what collision-resistance is, the logical question is: What is collision-resistance useful for? Here's one application: If we know that two inputs x and y to a collision-resistant hash function ***H*** have different hashes, then it's safe to assume that *x* and *y* are different — if someone knew an *x* and *y* that were different but had the same hash, that would violate our assumption that ***H*** is collision resistant.

This argument allows us to use hash outputs as a ***message digest***. Consider SecureBox, an authenticated online file storage system that allows users to upload files and ensure their integrity when they download them. Suppose that Alice uploads really large file, and wants to be able to verify later that the file she downloads is the same as the one she uploads. One way to do that would be to save the whole big file locally, and directly compare it to the file she downloads. While this works, it largely defeats the purpose of uploading it in the first place; if Alice needs to have access to a local copy of the file to ensure its integrity, she can just use the local copy directly.

Collision-free hashes provide an elegant and efficient solution to this problem. Alice just needs to remember the hash of the original file. When she later downloads the file from SecureBox, she computes the hash of the downloaded file and compares it to the one she stored. If the hashes are the same, then she can conclude that the file is indeed the one she uploaded, but if they are different, then Alice can conclude that the file has been tampered with. Remembering the hash thus allows her to detect *accidental* corruption of the file during transmission or on SecureBox's servers, but also *intentional* modification of the file by the server. Such guarantees in the face of potentially malicious behavior by other entities are at the core of what cryptography gives us.

The hash serves as a fixed length digest, or unambiguous summary, of a message. This gives us a very efficient way to remember things we've seen before and recognize them again. Whereas the entire file might have been gigabytes long, the hash is of fixed length, 256-bits for the hash function in our example. This greatly reduces our storage requirement. Later in this chapter and throughout the book, we'll see applications for which it's useful to use a hash as a message digest.

Property 2: Hiding The second property that we want from our hash functions is that it's ***hiding***. The hiding property asserts that if we're given the output of the hash function $y = H(x)$, there's no feasible way to figure out what the input, x, was. The problem is that this property can't be true in the stated form. Consider the following simple example: we're going to do an experiment where we flip a coin. If the result of the coin flip was heads, we're going to announce the hash of the string "heads". If the result was tails, we're going to announce the hash of the string "tails".

We then ask someone, an adversary, who didn't see the coin flip, but only saw this hash output, to figure out what the string was that was hashed (we'll soon see why we might want to play games like this). In response, they would simply compute both the hash of the string "heads" and the hash of the string "tails", and they could see which one they were given. And so, in just a couple steps, they can figure out what the input was.

The adversary was able to guess what the string was because there were only two possible values of x, and it was easy for the adversary to just try both of them. In order to be able to achieve the hiding property, it needs to be the case that there's no value of x which is particularly likely. That is, x has to be chosen from a set that's, in some sense, very spread out. If x is chosen from such a set, this method of trying a few values of x that are especially likely will not work.

The big question is: can we achieve the hiding property when the values that we want do not come from a spread out set as in our "heads" and "tails" experiment? Fortunately, the answer is yes! So perhaps we can hide even an input that's not spread out by concatenating it with another input that *is* spread out. We can now be slightly more precise about what we mean by hiding (the double vertical bar $\|$ denotes concatenation).

Hiding. A hash function H is hiding if: when a secret value r is chosen from a probability distribution that has *high min-entropy*, then given $H(r \| x)$ it is infeasible to find x.

In information-theory, ***min-entropy*** is a measure of how predictable an outcome is, and high min-entropy captures the intuitive idea that the distribution (i.e., random variable) is very spread out. What that means specifically is that when we sample from the distribution, there's no particular value that's likely to occur. So, for a concrete example, if r is chosen uniformly from among all of the strings that are 256 bits long, then any particular string was chosen with probability 1/2^256, which is an infinitesimally small value.

Application: Commitments. Now let's look at an application of the hiding property. In particular, what we want to do is something called a ***commitment***. A commitment is the digital analog of taking a value, sealing it in an envelope, and putting that envelope out on the table where everyone can see it. When you do that, you've committed yourself to what's inside the envelope. But you haven't opened it, so even though you've committed to a value, the value remains a secret from everyone else. Later, you can open the envelope and reveal the value that you committed to earlier.

Commitment scheme. A commitment scheme consists of two algorithms:

- **(*com*) := commit(*msg, key*)** The commit function takes a message and secret key as input and returns a commitment.
- ***isValid* := verify(*com, key, msg*)** The verify function takes a commitment, key, and message as input. It returns true if the *com* is a valid commitment to *msg* under the key, *key*. It returns false otherwise.

We require that the following two security properties hold:

- ***Hiding***: Given *com*, it is infeasible to find *msg*
- ***Binding***: For any value of *key*, it is infeasible to find two messages, *msg* and *msg'* such that *msg* ≠ *msg'* and verify(commit(*msg, key*), *key, msg'*) == true

To use a commitment scheme, one commits to a value, and publishes the commitment *com*. This stage is analogous to putting the sealed envelope on the table. At a later point, if they want to reveal the value that they committed to earlier, they publish the key, *key* and the value, *msg*. Now, anybody can verify that *msg* was indeed the value committed to earlier. This stage is analogous to opening up the envelope.

The two security properties dictate that the algorithms actually behave like sealing and opening an envelope. First, given *com*, the commitment, someone looking at the envelope can't figure out what the message is. The second property is that it's binding. That when you commit to what's in the envelope, you can't change your mind later. That is, it's infeasible to find two different messages, such that you can commit to one message, and then later claim that you committed to another.

So how do we know that these two properties hold? Before we can answer this, we need to discuss how we're going to actually implement a commitment scheme. We can do so using a cryptographic hash function. Consider the following commitment scheme:

- commit(*msg)* := (H(*key* ‖ *msg*), *key*)
    - where *key* is a random 256-bit value
- verify(*com, key, msg*) := **true** if H(*key* ‖ *msg) = com*; **false** otherwise

In this scheme, to commit to a value, we generate a random 256-bit value, which will serve as the key. And then we return the hash of the key concatenated together with the message as the commitment. To verify, someone is going to compute this same hash of the key they were given concatenated with the message. And they're going to check whether that's equal to the commitment that they saw.

Take another look at the two properties that we require of our commitment schemes. If we substitute the instantiation of *commit* and *verify* as well as *H(key* ‖ *msg)* for *com*, then these properties become:

- ***Hiding***: Given H(*key* ‖ *msg)*, infeasible to find *msg*
- ***Binding***: For any value of *key*, it is infeasible to find two messages, *msg* and *msg'* such that *msg* ≠ *msg'* and H(*key* ‖ *msg)* = H(*key* ‖ *msg')*

The *hiding* property of commitments is exactly the hiding property that we required for our hash functions. If *key* was chosen as a random 256-bit value then the hiding property says that if we hash the concatenation of *key* and the message, then it's infeasible to recover the message from the hash output. And it turns out that the *binding property* is implied by[1] the collision-resistant property of the underlying hash function. If the hash function is collision-resistant, then it will be infeasible to find distinct values msg and msg' such that $H(key \parallel msg) = H(key \parallel msg')$ since such values would indeed be a collision.

Therefore, if *H* is a hash function that is collision-resistant and hiding, this commitment scheme will work, in the sense that it will have the necessary security properties.

Property 3: Puzzle friendliness. The third security property we're going to need from hash functions is that they are puzzle-friendly. This property is a bit complicated. We will first explain what the technical requirements of this property are and then give an application that illustrates why this property is useful.

> ***Puzzle friendliness.*** A hash function *H* is said to be puzzle-friendly if for every possible n-bit output value *y*, if k is chosen from a distribution with high min-entropy, then it is infeasible to find *x* such that $H(k \parallel x) = y$ in time significantly less than 2^n.

Intuitively, what this means is that if someone wants to target the hash function to come out to some particular output value *y*, that if there's part of the input that is chosen in a suitably randomized way, it's very difficult to find another value that hits exactly that target.

Application: Search puzzle. Now, let's consider an application that illustrates the usefulness of this property. In this application, we're going to build a ***search puzzle***, a mathematical problem which requires searching a very large space in order to find the solution. In particular, a search puzzle has no shortcuts. That is, there's no way to find a valid solution other than searching that large space.

> ***Search puzzle.*** A search puzzle consists of
>
> - a hash function, *H*,
> - a value, *id* (which we call the ***puzzle-ID***), chosen from a high min-entropy distribution
> - and a target set *Y*
>
> A solution to this puzzle is a value, *x*, such that
>
> $$H(id \parallel x) \in \boldsymbol{Y}.$$

---

[1] The reverse implications do not hold. That is, it's possible that you cannot find a collision of the form $H(key \parallel msg) == H(key \parallel msg')$, but some other collision does exist.

The intuition is this: if H has an n-bit output, then it can take any of 2^n values. Solving the puzzle requires finding an input so that the output falls within the set Y, which is typically much smaller than the set of all outputs. The size of Y determines how hard the puzzle is. If Y is the set of all n-bit strings the puzzle is trivial, whereas if Y has only 1 element the puzzle is maximally hard. The fact that the puzzle id has high min-entropy ensures that there are no shortcuts. On the contrary, if a particular value of the ID were likely, then someone could cheat, say by pre-computing a solution to the puzzle with that ID.

If a search puzzle is puzzle-friendly, this implies that there's no solving strategy for this puzzle which is much better than just trying random values of *x*. And so, if we want to pose a puzzle that's difficult to solve, we can do it this way as long as we can generate puzzle-IDs in a suitably random way. We're going to use this idea later when we talk about Bitcoin mining, which is a sort of computational puzzle.

SHA-256. We've discussed three properties of hash functions, and one application of each of those. Now let's discuss a particular hash function that we're going to use a lot in this book. There are lots of hash functions in existence, but this is the one Bitcoin uses primarily, and it's a pretty good one to use. It's called ***SHA-256***.

Recall that we require that our hash functions work on inputs of arbitrary length. Luckily, as long as we can build a hash function that works on fixed-length inputs, there's a generic method to convert it into a hash function that works on arbitrary-length inputs. It's called the ***Merkle-Damgard transform***. SHA-256 is one of a number of commonly used hash functions that make use of this method. In common terminology, the underlying fixed-length collision-resistant hash function is called the ***compression function***. It has been proven that if the underlying compression function is collision resistant, then the overall hash function is collision resistant as well.

The Merkle-Damgard transform is quite simple. Say the compression function takes inputs of length *m* and produces an output of a smaller length *n*. The input to the hash function, which can be of any size, is divided into ***blocks*** of length *m-n*. The construction works as follows: pass each block together with the output of the previous block into the compression function. Notice that input length will then be $(m-n) + n = m$, which is the input length to the compression function. For the first block, to which there is no previous block output, we instead use an ***Initialization Vector (IV)***. This number is reused for every call to the hash function, and in practice you can just look it up in a standards document. The last block's output is the result that you return.

SHA-256 uses a compression function that takes 768-bit input and produces 256-bit outputs. The block size is 512 bits. See Figure 1.3 for a graphical depiction of how SHA-256 works.

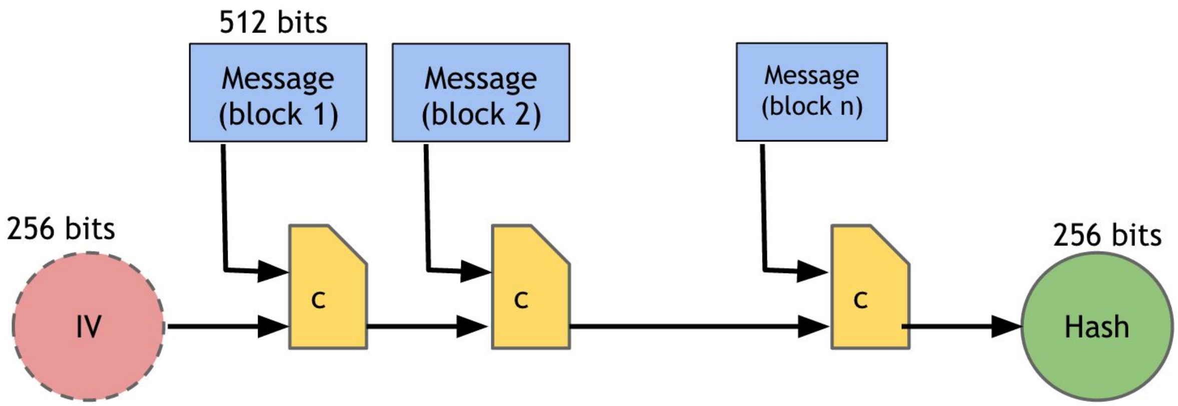

Figure 1.3: SHA-256 Hash Function (simplified). SHA-256 uses the Merkle-Damgard transform to turn a fixed-length collision-resistant compression function into a hash function that accepts arbitrary length inputs.

We've talked about hash functions, cryptographic hash functions with special properties, applications of those properties, and a specific hash function that we use in Bitcoin. In the next section, we'll discuss ways of using hash functions to build more complicated data structures that are used in distributed systems like Bitcoin.

1.2 Hash Pointers and Data Structures

In this section, we're going to discuss ***hash pointers*** and their applications. A hash pointer is a data structure that turns out to be useful in many of the systems that we will talk about. A hash pointer is simply a pointer to where some information is stored together with a cryptographic hash of the information. Whereas a regular pointer gives you a way to retrieve the information, a hash pointer also gives you a way to verify that the information hasn't changed.

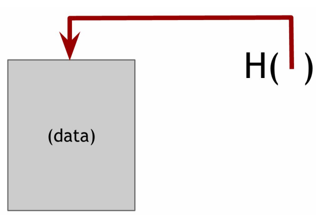

Figure 1.4 Hash pointer. A hash pointer is a pointer to where data is stored together with a cryptographic hash of the value of that data at some fixed point in time.

We can use hash pointers to build all kinds of data structures. Intuitively, we can take a familiar data structure that uses pointers such as a linked list or a binary search tree and implement it with hash pointers, instead of pointers as we normally would.

Block chain. In Figure 1.5, we built a linked list using hash pointers. We're going to call this data structure a ***block chain***. Whereas as in a regular linked list where you have a series of blocks, each block has data as well as a pointer to the previous block in the list, in a block chain the previous block pointer will be replaced with a hash pointer. So each block not only tells us where the value of the previous block was, but it also contains a digest of that value that allows us to verify that the value hasn't changed. We store the head of the list, which is just a regular hash-pointer that points to the most recent data block.

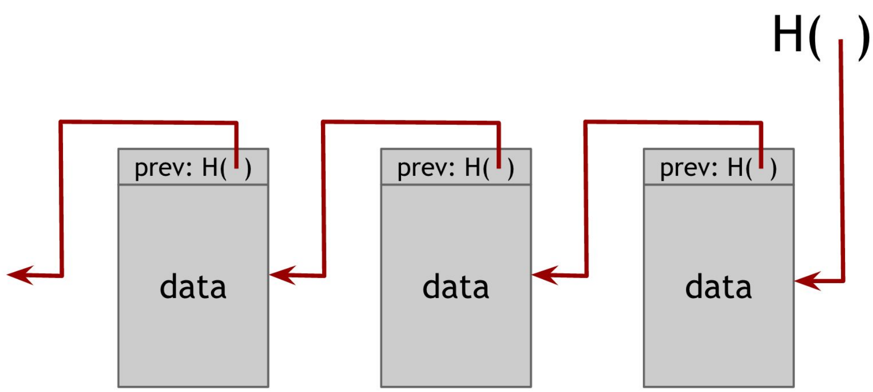

Figure 1.5 Block chain. A block chain is a linked list that is built with hash pointers instead of pointers.

A use case for a block chain is a ***tamper-evident log***. That is, we want to build a log data structure that stores a bunch of data, and allows us to append data onto the end of the log. But if somebody alters data that is earlier in the log, we're going to detect it.

To understand why a block chain achieves this tamper-evident property, let's ask what happens if an adversary wants to tamper with data that's in the middle of the chain. Specifically, the adversary's goal is to do it in such a way that someone who remembers only the hash pointer at the head of the block chain won't be able to detect the tampering. To achieve this goal, the adversary changes the data of some block k. Since the data has been changed, the hash in block k + 1, which is a hash of the entire block k, is not going to match up. Remember that we are statistically guaranteed that the new hash will not match the altered content since the hash function is collision resistant. And so we will detect the inconsistency between the new data in block k and the hash pointer in block k + 1. Of course the adversary can continue to try and cover up this change by changing the next block's hash as well. The adversary can continue doing this, but this strategy will fail when he reaches the head of the list. Specifically, as long as we store the hash pointer at the head of the list in a place where the adversary cannot change it, the adversary will be unable to change any block without being detected.

The upshot of this is that if the adversary wants to tamper with data anywhere in this entire chain, in order to keep the story consistent, he's going to have to tamper with the hash pointers all the way back to the beginning. And he's ultimately going to run into a roadblock because he won't be able to tamper with the head of the list. Thus it emerges, that by just remembering this single hash pointer, we've essentially remembered a tamper-evident hash of the entire list. So we can build a block chain like this containing as many blocks as we want, going back to some special block at the beginning of the list, which we will call the ***genesis block***.

You may have noticed that the block chain construction is similar to the Merkle-Damgard construction that we saw in the previous section. Indeed, they are quite similar, and the same security argument applies to both of them.

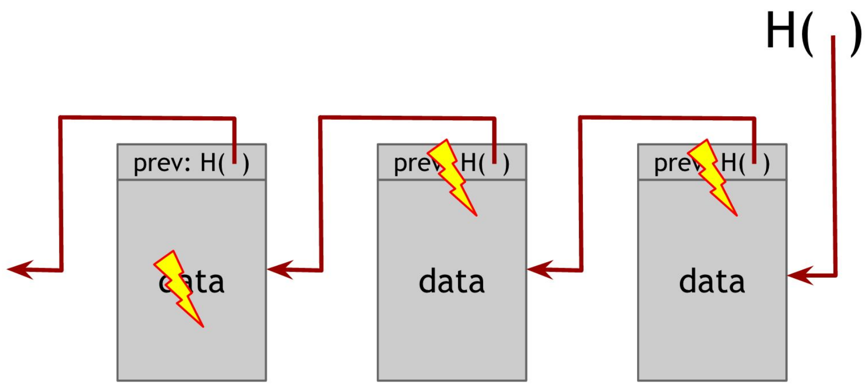

Figure 1.6 Tamper-evident log. If an adversary modifies data anywhere in the block chain, it will result in the hash pointer in the following block being incorrect. If we store the head of the list, then even if the adversary modifies all of the pointers to be consistent with the modified data, the head pointer will be incorrect, and we will detect the tampering.

Merkle trees. Another useful data structure that we can build using hash pointers is a binary tree. A binary tree with hash pointers is known as a ***Merkle tree***, after its inventor Ralph Merkle. Suppose we have a number of blocks containing data. These blocks comprise the leaves of our tree. We group these data blocks into pairs of two, and then for each pair, we build a data structure that has two hash pointers, one to each of these blocks. These data structures make the next level up of the tree. We in turn group these into groups of two, and for each pair, create a new data structure that contains the hash of each. We continue doing this until we reach a single block, the root of the tree.

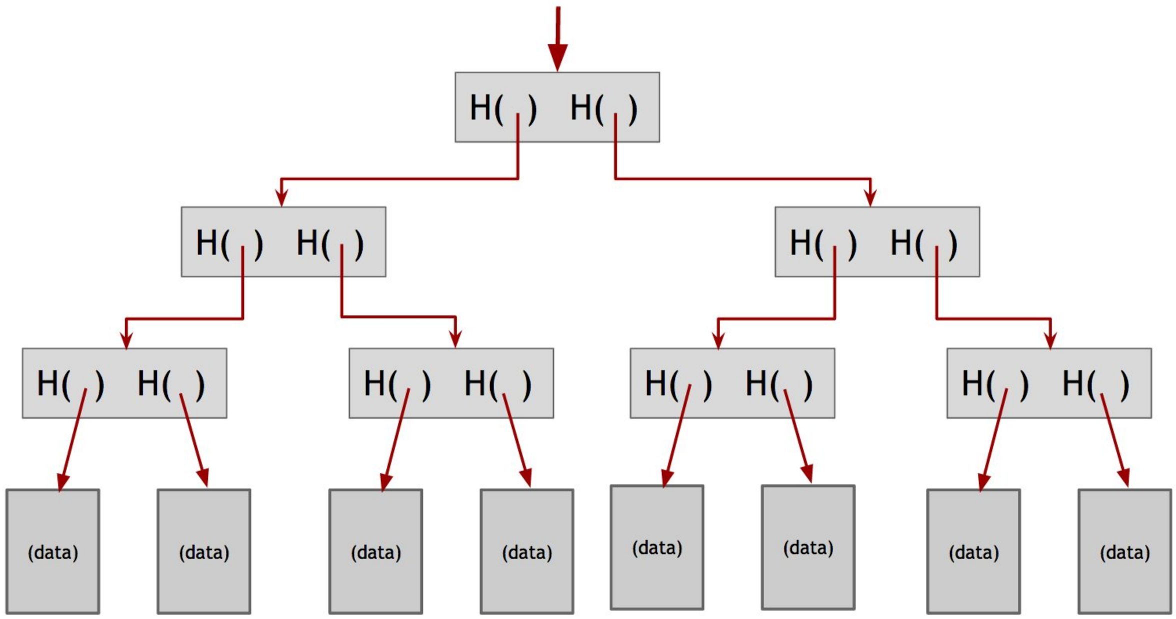

Figure 1.7 Merkle tree. In a Merkle tree, data blocks are grouped in pairs and the hash of each of these blocks is stored in a parent node. The parent nodes are in turn grouped in pairs and their hashes stored one level up the tree. This continues all the way up the tree until we reach the root node.

As before, we remember just the hash pointer at the head of the tree. We now have the ability traverse down through the hash pointers to any point in the list. This allows us make sure that the data hasn't been tampered with because, just like we saw with the block chain, if an adversary tampers with some data block at the bottom of the tree, that will cause the hash pointer that's one level up to not match, and even if he continues to tamper with this block, the change will eventually propagate to the top of the tree where he won't be able to tamper with the hash pointer that we've stored. So again, any attempt to tamper with any piece of data will be detected by just remembering the hash pointer at the top.

Proof of membership. Another nice feature of Merkle trees is that, unlike the block chain that we built before, it allows a concise proof of membership. Say that someone wants to prove that a certain data block is a member of the Merkle Tree. As usual, we remember just the root. Then they need to show us this data block, and the blocks on the path from the data block to the root. We can ignore the rest of the tree, as the blocks on this path are enough to allow us to verify the hashes all the way up to the root of the tree. See Figure 1.8 for a graphical depiction of how this works.

If there are n nodes in the tree, only about *log(n)* items need to be shown. And since each step just requires computing the hash of the child block, it takes about *log(n)* time for us to verify it. And so even if the Merkle tree contains a very large number of blocks, we can still prove membership, in a relatively short time. Verification thus runs in time and space that are logarithmic in the number of nodes in the tree.

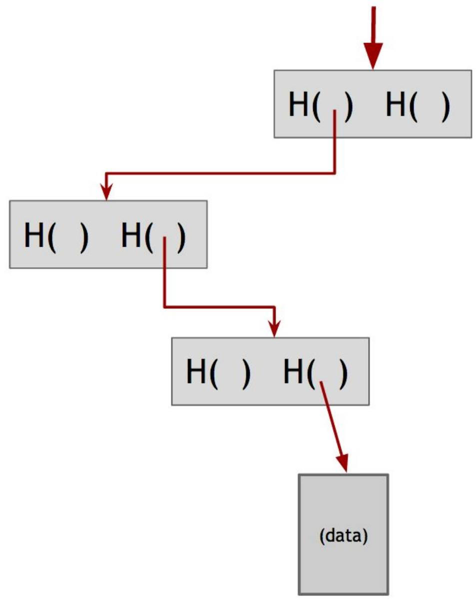

Figure 1.8 Proof of membership. To prove that a data block is included in the tree, one only needs to show the blocks in the path from that data block to the root.

A ***sorted Merkle tree*** is just a Merkle tree where we take the blocks at the bottom, and we sort them using some ordering function. This can be alphabetical, lexicographical order, numerical order, or some other agreed upon ordering.

Proof of non-membership. With a sorted Merkle tree, it becomes possible to verify non-membership in a logarithmic time and space. That is, we can prove that a particular block is not in the Merkle tree. And the way we do that is simply by showing a path to the item that's just before where the item in question would be and showing the path to the item that is just after where it would be. If these two items are consecutive in the tree, then this serves as a proof that the item in question is not included. For if it was included, it would need to be between the two items shown, but there is no space between them as they are consecutive.

We've discussed using hash pointers in linked lists and binary trees, but more generally, it turns out that we can use hash pointers in any pointer-based data structure as long as the data structure doesn't have cycles. If there are cycles in the data structure, then we won't be able to make all the hashes match up. If you think about it, in an acyclic data structure, we can start near the leaves, or near the things that don't have any pointers coming out of them, compute the hashes of those, and then work our way back toward the beginning. But in a structure with cycles, there's no end we can start with and compute back from.

So, to consider another example, we can build a directed acyclic graph out of hash pointers. And we'll be able to verify membership in that graph very efficiently. And it will be easy to compute. Using hash pointers in this manner is a general trick that you'll see time and again in the context of the distributed data structures and throughout the algorithms that we discuss later in this chapter and throughout this book.

1.3 Digital Signatures

In this section, we'll look at ***digital signatures***. This is the second cryptographic primitive, along with hash functions, that we need as building blocks for the cryptocurrency discussion later on. A digital signature is supposed to be the digital analog to a handwritten signature on paper. We desire two properties from digital signatures that correspond well to the handwritten signature analogy. Firstly, only you can make your signature, but anyone who sees it can verify that it's valid. Secondly, we want the signature to be tied to a particular document so that the signature cannot be used to indicate your agreement or endorsement of a different document. For handwritten signatures, this latter property is analogous to assuring that somebody can't take your signature and snip it off one document and glue it onto the bottom of another one.

How can we build this in a digital form using cryptography? First, let's make the previous intuitive discussion slightly more concrete. This will allow us to reason better about digital signature schemes and discuss their security properties.

Digital signature scheme. A digital signature scheme consists of the following three algorithms:

- **(sk, pk) := generateKeys(*keysize*)** The generateKeys method takes a key size and generates a key pair. The secret key *sk* is kept privately and used to sign messages. *pk* is the public verification key that you give to everybody. Anyone with this key can verify your signature.
- **sig := sign(*sk*, *message*)** The sign method takes a message, *msg*, and a secret key, *sk*, as input and outputs a signature for the *msg* under *sk*
- **isValid := verify(*pk*, *message*, *sig*)** The verify method takes a message, a signature, and a public key as input. It returns a boolean value, *isValid*, that will be ***true*** if *sig* is a valid signature for *message* under public key *pk*, and ***false*** otherwise.

We require that the following two properties hold:

- *Valid signatures must verify*
  - **verify**(*pk*, *message*, **sign**(*sk*, *message*)) == **true**
- Signatures are ***existentially unforgeable***

We note that **generateKeys** and **sign** can be randomized algorithms. Indeed, generateKeys had better be randomized because it ought to be generating different keys for different people. **verify**, on the other hand, will always be deterministic.

Let us now examine the two properties that we require of a digital signature scheme in more detail. The first property is straightforward — that valid signatures must verify. If I sign a message with *sk*, my secret key, and someone later tries to validate that signature over that same message using my public