CHAPTER 1

Introduction to Cryptography and 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 anticounterfeiting 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, in a mathematical protocol, the rules for creation of new units of the currency. Thus, before we can properly understand cryptocurrencies, we need to delve into the cryptographic foundations that they rely on.

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

Once the necessary cryptographic primitives have been introduced, we'll discuss some of the ways in which they are used to build cryptocurrencies. We'll complete this chapter with examples of simple cryptocurrencies that illustrate some of the design challenges that need to be dealt with.

1.1. CRYPTOGRAPHIC HASH FUNCTIONS

The first cryptographic primitive that we 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-sized 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 $O(n)$.

These 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 require that it has the following three additional properties: (1) collision resistance, (2) hiding, and (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 satisfies them. The reader who has studied cryptography should be aware that the treatment of hash functions in this book is a bit different from that in 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 is collision resistant. A collision occurs when two distinct inputs produce the same output. A hash function $H(\cdot)$ is collision resistant if nobody can find a collision (Figure 1.1). 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 $H(x) = H(y)$.

Notice that we said "nobody can find" a collision, but we did not say that no collisions exist. Actually, collisions exist for any hash function, 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

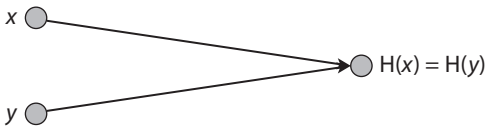

FIGURE 1.1. A hash collision. *x* and *y* are distinct values, yet when input into hash function *H*, they produce the same output.

string. In fact, there will be some outputs to which an infinite number of possible inputs will map (Figure 1.2).

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 whether 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 percent chance that at least two of them are going to collide. That we can find a collision by examining only 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 (see the online supplementary material for this book, which can be found at http://press.princeton.edu/titles/10908.html), we examine this in more detail.

This collision-detection algorithm works for every hash function. But, of course, the problem is that it takes a 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

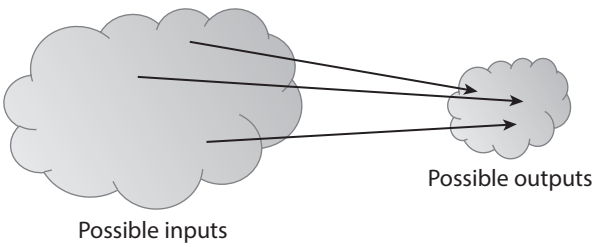

FIGURE 1.2. Inevitability of collisions. Because the number of inputs exceeds the number of outputs, we are guaranteed that there must be at least one output to which the hash function maps more than one input.

(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 had been computing since the beginning of the universe, the odds that they would have found a collision by now are still infinitesimally small. So small that it's far less than the odds that the Earth will be destroyed by a giant meteor in the next two seconds.

We have thus found 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 to find a collision? In other words, although the generic collision detection algorithm is not feasible to use, there 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 whether such methods exist. We suspect that they are collision resistant. However, no hash functions have been *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. And so we choose to believe that those are collision resistant. (In some cases, such as the hash function known as MD5, collisions were eventually found after years of work, resulting in the function being deprecated and phased out of practical use.)

APPLICATION: MESSAGE DIGESTS

Now that we know what collision resistance is, the logical question is: What is it useful for? Here's one application: If we know that two inputs x and y to a collision-resistant hash function H are different, then it's safe to assume that their hashes $H(x)$ and $H(y)$ 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 to ensure their integrity when they download them. Suppose that Alice uploads really large files, and she wants to be able to verify later that the file she downloads is the same as the one she uploaded. 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-resistant 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 same 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 not only 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 to 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 is *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 form stated. 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 only two values of x were possible, and it was easy for the adversary to just try both of them. To be able to achieve the hiding property, there must be no value of x that is particularly likely. That is, x has to be chosen from a set that is, 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! 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 said to be 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., of a 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 strings that are 256 bits long, then any particular string is 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*, *nonce*) The commit function takes a message and secret random value, called a *nonce*, as input and returns a commitment.
- verify(*com*, *msg*, *nonce*) The verify function takes a commitment, nonce, and message as input. It returns true if *com* == commit(*msg, nonce*) and false otherwise.

We require that the following two security properties hold:

- *Hiding*: Given *com*, it is infeasible to find *msg*.
- *Binding*: It is infeasible to find two pairs (*msg, nonce*) and (*msg′*, *nonce′*) such that *msg* ≠ *msg′* and commit(*msg, nonce*) == commit(*msg′*, *nonce′*).

To use a commitment scheme, we first need to generate a random *nonce*. We then apply the *commit* function to this nonce together with *msg*, the value being committed

to, and we publish the commitment *com*. This stage is analogous to putting the sealed envelope on the table. At a later point, if we want to reveal the value that we committed to earlier, we publish the random nonce that we used to create this commitment, and the message, *msg*. Now anybody can verify that *msg* was indeed the message committed to earlier. This stage is analogous to opening the envelope.

Every time you commit to a value, it is important that you choose a new random value *nonce*. In cryptography, the term *nonce* is used to refer to a value that can only be used once.

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. This ensures 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:

$$\text{commit}(msg, nonce) := H(nonce \parallel msg),$$
where *nonce* is a random 256-bit value

To commit to a message, we generate a random 256-bit nonce. Then we concatenate the nonce and the message and return the hash of this concatenated value as the commitment. To verify, someone will compute this same hash of the nonce they were given concatenated with the message. And they will check whether the result is equal to the commitment that they saw.

Take another look at the two properties required of our commitment schemes. If we substitute the instantiation of *commit* and *verify* as well as $H(nonce \parallel msg)$ for *com*, then these properties become:

- *Hiding*: Given $H(nonce \parallel msg)$, it is infeasible to find *msg*.
- *Binding*: It is infeasible to find two pairs $(msg, nonce)$ and $(msg', nonce')$ such that $msg \neq msg'$ and $H(nonce \parallel msg) == (nonce' \parallel msg')$.

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 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 $H(nonce \parallel msg) = H(nonce' \parallel msg')$, since such values would indeed be a collision. (Note that the reverse implications do not hold. That is, it's possible that you can find collisions, but none of them are of the form $H(nonce \parallel msg) == H(nonce' \parallel msg')$. For example, if you can only find a collision in which two distinct nonces generate the same commitment for the same message, then the commitment scheme is still binding, but the underlying hash function is not collision resistant.)

Therefore, if H is a hash function that is both 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 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, if someone wants to target the hash function to have some particular output value y, and if part of the input has been chosen in a suitably randomized way, then it's very difficult to find another value that hits exactly that target.

APPLICATION: SEARCH PUZZLE

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 that requires searching a very large space 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

$$H(id \parallel x) \in Y.$$

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 such 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, then the puzzle is trivial, whereas if Y has only one element, then the puzzle is maximally hard. 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 precomputing a solution to the puzzle with that ID.

If a hash funtion is puzzle friendly, then there's no solving strategy for this puzzle that 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, starting in Chapter 2—mining is a sort of computational puzzle.

SHA-256

We've discussed three properties of hash functions and one application of each of these properties. Now let's discuss a particular hash function that we're going to use a lot in this book. Many hash functions exist, 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-Damgård 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-Damgård transform is quite simple. Suppose that 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 in Figure 1.3). 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.

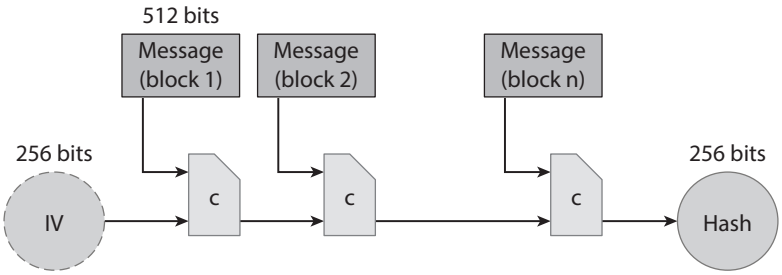

FIGURE 1.3. SHA-256 hash function (simplified). SHA-256 uses the Merkle-Damgård transform to turn a fixed-length collision-resistant compression function into a hash function that accepts arbitrary-length inputs. The input is padded, so that its length is a multiple of 512 bits. IV stands for initialization vector.

Modeling Hash Functions

Hash functions are the Swiss Army knife of cryptography: they find a place in a spectacular variety of applications. The flip side to this versatility is that different applications require slightly different properties of hash functions to ensure security. It has proven notoriously hard to pin down a list of hash function properties that would result in provable security across the board.

In this text, we've selected three properties that are crucial to the way that hash functions are used in Bitcoin and other cryptocurrencies. Even in this space, not all of these properties are necessary for every use of hash functions. For example, puzzle friendliness is only important in Bitcoin mining, as we'll see.

Designers of secure systems often throw in the towel and model hash functions as functions that output an independent random value for every possible input. The use of this "random oracle model" for proving security remains controversial in cryptography. Regardless of one's position on this debate, reasoning about how to reduce the security properties that we want in our applications to fundamental properties of the underlying primitives is a valuable intellectual exercise for building secure systems. Our presentation in this chapter is designed to help you learn this skill.

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.

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 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 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 consider. A hash

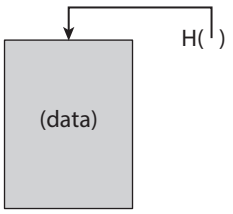

FIGURE 1.4. Hash pointer. A hash pointer is a pointer to where data is stored together with a cryptographic hash of the value of this data at some fixed point in time.

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 allows you to verify that the information hasn't been changed (Figure 1.4).

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 ordinary pointers, as we normally would.

Block Chain

Figure 1.5 shows a linked list using hash pointers. We call this data structure a *block chain*. 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. But 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, which allows us to verify that the value hasn't been changed. We store the head of the list, which is just a regular hash-pointer that points to the most recent data block.

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

To understand why a block chain achieves this tamper-evident property, let's ask what happens if an adversary wants to tamper with data in the middle of the chain.

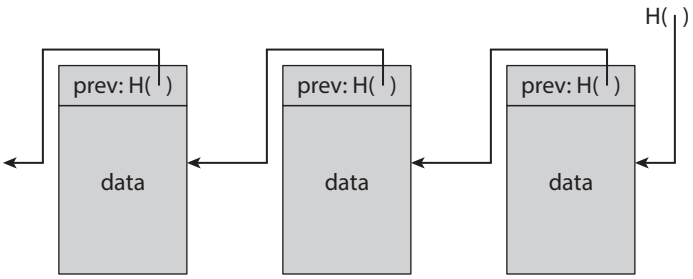

FIGURE 1.5. Block chain. A block chain is a linked list that is built with hash pointers instead of pointers.

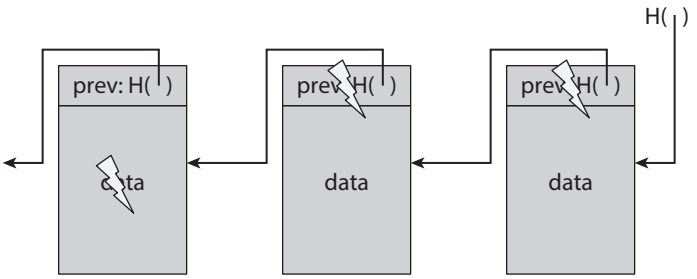

FIGURE 1.6. Tamper-evident log. If an adversary modifies data anywhere in the block chain, it will result in the hash pointer in the following block being incorrect. If we store the head of the list, then even if an adversary modifies all pointers to be consistent with the modified data, the head pointer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

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 she 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, she will be unable to change any block without being detected (Figure 1.6).

The upshot is that if the adversary wants to tamper with data anywhere in this entire chain, to keep the story consistent, she's going to have to tamper with the hash pointers all the way to the end. And she's ultimately going to run into a roadblock, because she won't be able to tamper with the head of the list. Thus, by remembering just this single hash pointer, we've essentially determined 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-Damgård construction discussed in Section 1.1. Indeed, they are quite similar, and the same security argument applies to both of them.

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* (Figure 1.7), after its inventor, Ralph Merkle. Suppose we have some blocks containing data. These blocks make up the

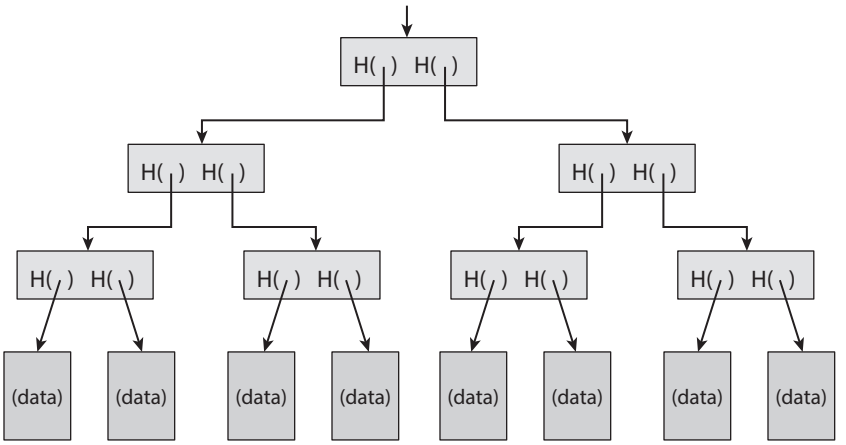

FIGURE 1.7. Merkle tree. In a Merkle tree, data blocks are grouped in pairs, and the hash of each of these blocks is stored in a parent node. The parent nodes are in turn grouped in pairs, and their hashes stored one level up the tree. This pattern continues up the tree until we reach the root node.

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 the blocks. These data structures make up the next level 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.

As before, we remember just one hash pointer: in this case, the one at the root of the tree. We now have the ability to traverse through the hash pointers to any point in the list. This allows us to make sure that the data has not been tampered with because, just as we saw for the block chain, if an adversary tampers with some data block at the bottom of the tree, his change will cause the hash pointer one level up to not match, and even if he continues to tamper with other blocks farther up the tree, the change will eventually propagate to the top, 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, they allow a concise *proof of membership*. Suppose 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

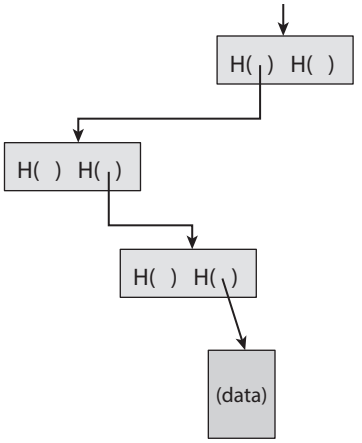

FIGURE 1.8. Proof of membership. To prove that a data block is included in the tree only requires showing the blocks in the path from that data block to the root.

for us to verify it. And so even if the Merkle tree contains a large number of blocks, we can still prove membership in a relatively short time. Verification thus runs in time and space that's logarithmic in the number of nodes in the tree.

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

Proof of Nonmembership

Using a sorted Merkle tree, it becomes possible to verify nonmembership in 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 just before where the item in question would be and showing the path to the item just after where it would be. If these two items are consecutive in the tree, then this serves as proof that the item in question is not included—because if it were 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 that we can start with and compute back from.

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. It also

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 distributed data structures and in the algorithms that we discuss later in this chapter (Section 1.5) and throughout the book.

1.3. DIGITAL SIGNATURES

In this section, we look at *digital signatures*. This is the second cryptographic primitive, along with hash functions, that we need as building blocks for the cryptocurrency discussion in Section 1.5. 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. First, only you can make your signature, but anyone who sees it can verify that it's valid. Second, 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 ensuring that somebody can't take your signature and snip it off one document and glue it to the bottom of another one.

How can we build this in a digital form using cryptography? First, let's make the above 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 and a secret key, *sk*, as input and outputs a signature for *message* 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. In contrast, verify will always be deterministic.

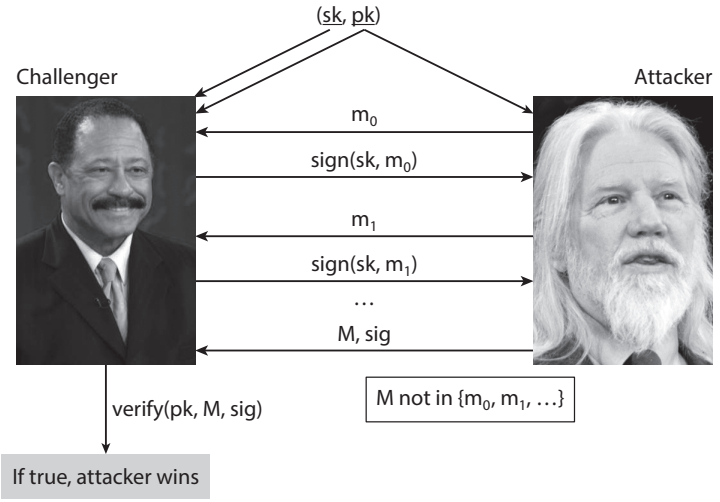

FIGURE 1.9. Unforgeability game. The attacker and the challenger play the unforgeability game. If the attacker is able to successfully output a signature on a message that he has not previously seen, he wins. If he is unable to do so, the challenger wins, and the digital signature scheme is unforgeable. Photograph of Whit Diffie (right), cropped, © Kevin Bocek. Licensed under Creative Commons CC BY 2.0.

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 be verifiable. 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 key, *pk*, the signature must validate correctly. This property is a basic requirement for signatures to be useful at all.

Unforgeability. The second requirement is that it's computationally infeasible to forge signatures. That is, an adversary who knows your public key and sees your signatures on some other messages can't forge your signature on some message for which he has not seen your signature. This unforgeability property is generally formalized in terms of a game that we play with an adversary. The use of games is quite common in cryptographic security proofs.

In the unforgeability game, an adversary claims that he can forge signatures, and a challenger tests this claim (Figure 1.9). The first thing we do is use generateKeys to generate a secret signing key and a corresponding public verification key. We give the secret key to the challenger, and we give the public key to both the challenger and the adversary. So the adversary only knows information that's public, and his mission is to try to forge a message. The challenger knows the secret key. So he can make signatures.

Intuitively, the setup of this game matches real-world conditions. A real attacker would likely be able to see valid signatures from his would-be victim on different documents. And the attacker might even be able to manipulate the victim into signing innocuous-looking documents if that's useful to the attacker.

To model this in our game, we allow the adversary to get signatures on some documents of his choice, for as long as he wants, as long as the number of guesses is plausible. To give an intuitive idea of what we mean by a plausible number of guesses, we would allow the adversary to try 1 million guesses, but not 2^{80} guesses. In asymptotic terms, we allow the adversary to try a number of guesses that is a polynomial function of the key size, but no more (e.g., he cannot try exponentially many guesses).

Once the adversary is satisfied that he's seen enough signatures, then he picks some message, *M*, that he will attempt to forge a signature on. The only restriction on *M* is that it must be a message for which the adversary has not previously seen a signature (because then he can obviously send back a signature that he has been given). The challenger runs the verify algorithm to determine whether the signature produced by the attacker is a valid signature on *M* under the public verification key. If it successfully verifies, the adversary wins the game.

We say that the signature scheme is unforgeable if and only if, no matter what algorithm the adversary is using, his chance of successfully forging a message is extremely small—so small that we can assume it will never happen in practice.

Practical Concerns

Several practical things must be done to turn the algorithmic idea into a digital signature mechanism that can be implemented. For example, many signature algorithms are randomized (in particular, the one used in Bitcoin), and we therefore need a good source of randomness. The importance of this requirement can't be overestimated, as bad randomness will make your otherwise-secure algorithm insecure.

Another practical concern is the message size. In practice, there's a limit on the message size that you're able to sign, because real schemes are going to operate on bit strings of limited length. There's an easy way around this limitation: sign the hash of the message, rather than the message itself. If we use a cryptographic hash function with a 256-bit output, then we can effectively sign a message of any length as long as our signature scheme can sign 256-bit messages. As we have discussed, it's safe to use the hash of the message as a message digest in this manner, since the hash function is collision resistant.

Another trick that we will use later is that you can sign a hash pointer. If you sign a hash pointer, then the signature covers, or protects, the whole structure—not just the hash pointer itself, but everything the chain of hash pointers points to. For example, if you were to sign the hash pointer located at the end of a block chain, the result is that you would effectively be digitally signing the entire block chain.

ECDSA

Now let's get into the nuts and bolts. Bitcoin uses a particular digital signature scheme known as the *Elliptic Curve Digital Signature Algorithm* (ECDSA). ECDSA is a U.S. government standard, an update of the earlier DSA algorithm adapted to use elliptic curves.

These algorithms have received considerable cryptographic analysis over the years and are generally believed to be secure.

More specifically, Bitcoin uses ECDSA over the standard elliptic curve secp256k1, which is estimated to provide 128 bits of security (i.e., it is as difficult to break this algorithm as it is to perform 2^{128} symmetric-key cryptographic operations, such as invoking a hash function). Although this curve is a published standard, it is rarely used outside Bitcoin; other applications using ECDSA (such as key exchange in the TLS protocol for secure web browsing) typically use the more common secp256r1 curve. This is just a quirk of Bitcoin, as it was chosen by Satoshi (see the Foreword) in the early specification of the system and is now difficult to change.

We won't go into all the details of how ECDSA works, as some complicated math is involved and understanding it is not necessary for the rest of this book. If you're interested in the details, refer to our Further Reading section at the end of this chapter. It might be useful to have an idea of the sizes of various quantities, however:

Private key:	256 bits
Public key, uncompressed:	512 bits
Public key, compressed:	257 bits
Message to be signed:	256 bits
Signature:	512 bits

Note that even though ECDSA can technically only sign messages 256 bits long, this is not a problem: messages are always hashed before being signed, so effectively any size message can be efficiently signed.

With ECDSA, a good source of randomness is essential, because a bad source will likely leak your key. It makes intuitive sense that if you use bad randomness when generating a key, then the key you generate will likely not be secure. But it's a quirk of ECDSA that, even if you use bad randomness only when making a signature and you use your perfectly good key, the bad signature will also leak your private key. (For those familiar with DSA, this is a general quirk in DSA and is not specific to the elliptic-curve variant.) And then it's game over: if you leak your private key, an adversary can forge your signature. We thus need to be especially careful about using good randomness in practice. Using a bad source of randomness is a common pitfall of otherwise secure systems.

This completes our discussion of digital signatures as a cryptographic primitive. In the next section, we discuss some applications of digital signatures that will turn out to be useful for building cryptocurrencies.

1.4. PUBLIC KEYS AS IDENTITIES

Let's look at a nice trick that goes along with digital signatures. The idea is to take a public key, one of those public verification keys from a digital signature scheme, and

Cryptocurrencies and Encryption

If you've been waiting to find out which encryption algorithm is used in Bitcoin, we're sorry to disappoint you. There is no encryption in Bitcoin, because nothing needs to be encrypted, as we'll see. Encryption is only one of a rich suite of techniques made possible by modern cryptography. Many of them, such as commitment schemes, involve hiding information in some way, but they are distinct from encryption.

equate it to an identity of a person or an actor in a system. If you see a message with a signature that verifies correctly under a public key, *pk*, then you can think of this as *pk* stating the message. You can literally think of a public key as being like an actor, or a party in a system, who can make statements by signing those statements. From this viewpoint, the public key is an identity. For someone to speak for the identity *pk*, he must know the corresponding secret key, *sk*.

A consequence of treating public keys as identities is that you can make a new identity whenever you want—you simply create a new fresh key pair, *sk* and *pk*, via the *generateKeys* operation in our digital signature scheme. This *pk* is the new public identity that you can use, and *sk* is the corresponding secret key that only you know and that lets you speak on behalf of the identity *pk*. In practice, you may use the hash of *pk* as your identity, since public keys are large. If you do that, then to verify that a message comes from your identity, one will have to check that (1) *pk* indeed hashes to your identity, and (2) the message verifies under public key *pk*.

Moreover, by default, your public key *pk* will basically look random, and nobody will be able to uncover your real-world identity by examining *pk*. (Of course, once you start making statements using this identity, these statements may leak information that allows others to connect *pk* to your real-world identity. We discuss this in more detail shortly.) You can generate a fresh identity that looks random, like a face in the crowd, and is controlled only by you.

Decentralized Identity Management

This brings us to the idea of decentralized identity management. Rather than having a central authority for registering users in a system, you can register as a user by yourself. You don't need to be issued a username, nor do you need to inform someone that you're going to be using a particular name. If you want a new identity, you can just generate one at any time, and you can create as many as you want. If you prefer to be known by five different names, no problem! Just make five identities. If you want to be somewhat anonymous for a while, you can create a new identity, use it for just a little while, and then throw it away. All these things are possible with decentralized identity management, and this is the way Bitcoin, in fact, handles identity. These identities are called *addresses*, in Bitcoin jargon. You'll frequently hear the term "address" used in the context of Bitcoin and cryptocurrencies, and it's really just a hash of a public key. It's an

Security and Randomness

The idea that you can generate an identity without a centralized authority may seem counterintuitive. After all, if someone else gets lucky and generates the same key as you, can't they steal your bitcoins?

The answer is that the probability of someone else generating the same 256-bit key as you is so small that we don't have to worry about it in practice. For all intents and purposes, we are guaranteed that it will never happen.

More generally, in contrast to beginners' intuition that probabilistic systems are unpredictable and hard to reason about, often the opposite is true—the theory of statistics allows us to precisely quantify the chances of events we're interested in and to make confident assertions about the behavior of such systems.

But there's a subtlety: the probabilistic guarantee is true only when keys are generated at random. The generation of randomness is often a weak point in real systems. If two users' computers use the same source of randomness or use predictable randomness, then the theoretical guarantees no longer apply. So to ensure that practical guarantees match the theoretical ones, it is crucial to use a good source of randomness when generating keys.

identity that someone made up out of thin air, as part of this decentralized identity management scheme.

At first glance, it may seem that decentralized identity management leads to great anonymity and privacy. After all, you can create a random-looking identity all by yourself without telling anyone your real-world identity. But it's not that simple. Over time, the identity that you create makes a series of statements. People see these statements and thus know that whoever owns this identity has done a certain series of actions. They can start to connect the dots, using this series of actions to make inferences about your real-world identity. An observer can link together these observations over time and make inferences that lead to such conclusions as, "Gee, this person is acting a lot like Joe. Maybe this person is Joe."

In other words, in Bitcoin you don't need to explicitly register or reveal your real-world identity, but the pattern of your behavior might itself be identifying. This is the fundamental privacy question in a cryptocurrency like Bitcoin, and indeed we'll devote Chapter 6 to it.

1.5. TWO SIMPLE CRYPTOCURRENCIES

Now let's move from cryptography to cryptocurrencies. Eating our cryptographic vegetables will start to pay off here, and we'll gradually see how the pieces fit together and why cryptographic operations like hash functions and digital signatures are actually useful. In this section we discuss two very simple cryptocurrencies. Of course, much of the rest of the book is needed to spell out all the details of how Bitcoin itself works.

Goofycoin

The first of the two is *Goofycoin*, which is about the simplest cryptocurrency we can imagine. There are just two rules of Goofycoin. The first rule is that a designated entity, Goofy, can create new coins whenever he wants and these newly created coins belong to him.

To create a coin, Goofy generates a unique coin ID `uniqueCoinID` that he's never generated before and constructs the string `CreateCoin [uniqueCoinID]`. He then computes the digital signature of this string with his secret signing key. The string, together with Goofy's signature, is a coin. Anyone can verify that the coin contains Goofy's valid signature of a `CreateCoin` statement and is therefore a valid coin.

The second rule of Goofycoin is that whoever owns a coin can transfer it to someone else. Transferring a coin is not simply a matter of sending the coin data structure to the recipient—it's done using cryptographic operations.

Let's say Goofy wants to transfer a coin that he created to Alice. To do this, he creates a new statement that says "Pay this to Alice" where "this" is a hash pointer that references the coin in question. And as we saw earlier, identities are really just public keys, so "Alice" refers to Alice's public key. Finally, Goofy signs the string representing the statement. Since Goofy is the one who originally owned that coin, he has to sign any transaction that spends the coin. Once this data structure representing Goofy's transaction is signed by him, Alice owns the coin. She can prove to anyone that she owns the coin, because she can present the data structure with Goofy's valid signature. Furthermore, it points to a valid coin that was owned by Goofy. So the validity and ownership of coins are self-evident in the system.

Once Alice owns the coin, she can spend it in turn. To do this, she creates a statement that says, "Pay this to Bob's public key" where "this" is a hash pointer to the coin that was owned by her. And of course, Alice signs this statement. Anyone, when presented with this coin, can verify that Bob is the owner. They can follow the chain of hash pointers back to the coin's creation and verify that at each step, the rightful owner signed a statement that says "pay this coin to [new owner]" (Figure 1.10).

To summarize, the rules of Goofycoin are:

- Goofy can create new coins by simply signing a statement that he's making a new coin with a unique coin ID.
- Whoever owns a coin can pass it on to someone else by signing a statement that says, "Pass on this coin to X" (where X is specified as a public key).
- Anyone can verify the validity of a coin by following the chain of hash pointers back to its creation by Goofy, verifying all signatures along the way.

Of course, there's a fundamental security problem with Goofycoin. Let's say Alice passed her coin on to Bob by sending her signed statement to Bob but didn't tell anyone else. She could create another signed statement that pays the same coin to Chuck. To

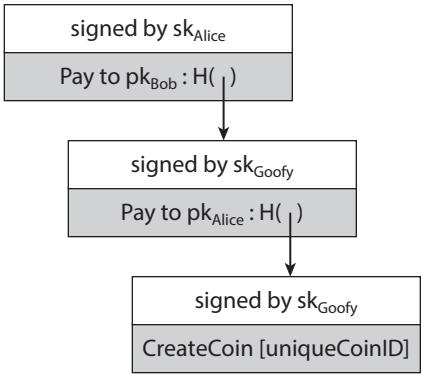

FIGURE 1.10. Goofycoin coin. Shown here is a coin that's been created (bottom) and spent twice (middle and top).

Chuck, it would appear that it is a perfectly valid transaction, and now he's the owner of the coin. Bob and Chuck would both have valid-looking claims to be the owner of this coin. This is called a *double-spending attack*—Alice is spending the same coin twice. Intuitively, we know coins are not supposed to work that way.

In fact, double-spending attacks are one of the key problems that any cryptocurrency has to solve. Goofycoin does not solve the double-spending attack, and therefore it's not secure. Goofycoin is simple, and its mechanism for transferring coins is actually similar to that of Bitcoin, but because it is insecure, it is inadequate as a cryptocurrency.

Scroogecoin

To solve the double-spending problem, we'll design another cryptocurrency, called *Scroogecoin*. Scroogecoin is built off of Goofycoin, but it's a bit more complicated in terms of data structures.

The first key idea is that a designated entity called Scrooge publishes an *append-only ledger* containing the history of all transactions. The append-only property ensures that any data written to this ledger will remain forever in the ledger. If the ledger is truly append only, we can use it to defend against double spending by requiring all transactions to be written in the ledger before they are accepted. That way, it will be publicly documented if coins were previously sent to a different owner.

To implement this append-only functionality, Scrooge can build a block chain (the data structure discussed in Section 1.2), which he will digitally sign. It consists of a series of data blocks, each with one transaction in it (in practice, as an optimization, we'd really put multiple transactions in the same block, as Bitcoin does.) Each block has the ID of a transaction, the transaction's contents, and a hash pointer to the previous block. Scrooge digitally signs the final hash pointer, which binds all the data in this entire structure, and he publishes the signature along with the block chain (Figure 1.11).

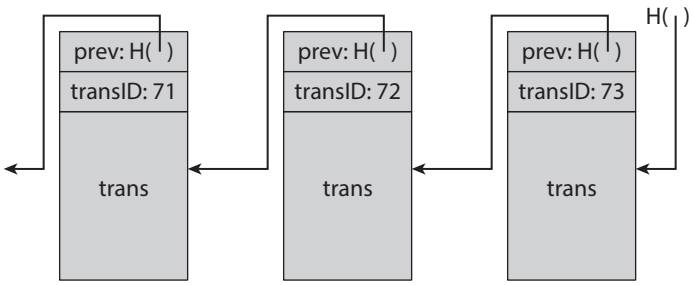

FIGURE 1.11. Scroogecoin block chain.

In Scroogecoin, a transaction only counts if it is in the block chain signed by Scrooge. Anybody can verify that a transaction was endorsed by Scrooge by checking Scrooge's signature on the block that records the transaction. Scrooge makes sure that he doesn't endorse a transaction that attempts to double spend an already spent coin.

Why do we need a block chain with hash pointers in addition to having Scrooge sign each block? This ensures the append-only property. If Scrooge tries to add or remove a transaction, or to change an existing transaction, it will affect all following blocks because of the hash pointers. As long as someone is monitoring the latest hash pointer published by Scrooge, the change will be obvious and easy to catch. In a system where Scrooge signed blocks individually, you'd have to keep track of every single signature Scrooge ever issued. A block chain makes it easy for any two individuals to verify that they have observed the same history of transactions signed by Scrooge.

In Scroogecoin, there are two kinds of transactions. The first kind is `CreateCoins`, which is just like the operation Goofy could do in Goofycoin to make a new coin. With Scroogecoin, we'll extend the semantics a bit to allow multiple coins to be created in one transaction (Figure 1.12).

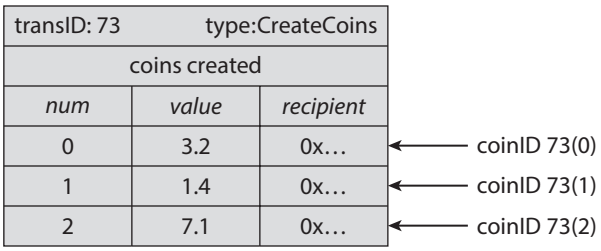

FIGURE 1.12. `CreateCoins` transaction. This `CreateCoins` transaction creates multiple coins. Each coin has a serial number in the transaction. Each coin also has a value; it's worth a certain number of scroogecoins. Finally, each coin has a recipient, which is a public key that gets the coin when it's created. So `CreateCoins` creates multiple new coins with different values and assigns them to people as initial owners. We refer to coins by CoinIDs. A CoinID is a combination of a transaction ID and the coin's serial number in that transaction.

transID: 73		type:PayCoins
consumed coinIDs: 68(1), 42(0), 72(3)		
coins created		
num	*value*	*recipient*
0	3.2	0x…
1	1.4	0x…
2	7.1	0x…
signatures		

FIGURE 1.13.A `PayCoins` transaction.

By definition, a `CreateCoins` transaction is always valid if it is signed by Scrooge. We won't worry about when or how many coins Scrooge is entitled to create, just like we didn't worry in Goofycoin about how Goofy was chosen as the entity allowed to create coins.

The second kind of transaction is `PayCoins`. It consumes some coins (i.e., destroys them) and creates new coins of the same total value. The new coins might belong to different people (public keys). This transaction has to be signed by everyone who's paying in a coin. So if you're the owner of one of the coins that's going to be consumed in this transaction, then you need to digitally sign the transaction to say that you're OK with spending this coin.

The rules of Scroogecoin say that the `PayCoins` transaction is valid if it satisfies four conditions:

- The consumed coins are valid, that is, they were created in previous transactions.
- The consumed coins have not already been consumed in some previous transaction. That is, this is not a double-spend transaction.
- The total value of the coins that come out of this transaction is equal to the total value of the coins that went in. That is, only Scrooge can create new value.
- The transaction is validly signed by the owners of all coins consumed in the transaction.

If these conditions are met, then this `PayCoins` transaction is valid, and Scrooge will accept it (Figure 1.13). He'll write it into the ledger by appending it to the block chain, after which everyone can see that this transaction has happened. It is only at this point that the participants can accept that the transaction has actually occurred. Until it is published, it might be preempted by a double-spending transaction even if it is otherwise validated by the first three conditions.

Coins in this system are immutable—they are never changed, subdivided, or com-

bined. Each coin is created, once, in one transaction and then later consumed in another transaction. But we can get the same effect as being able to subdivide or combine coins by using transactions. For example, to subdivide a coin, Alice creates a new transaction that consumes that one coin and then produces two new coins of the same total value. Those two new coins could be assigned back to her. So although coins are immutable in this system, it has all the flexibility of a system that doesn't have immutable coins.

Now we come to the core problem with Scroogecoin. Scroogecoin will work in the sense that people can see which coins are valid. It prevents double spending, because everyone can look into the block chain and see that all transactions are valid and that every coin is consumed only once. But the problem is Scrooge—he has too much influence. He can't create fake transactions, because he can't forge other people's signatures. But he could stop endorsing transactions from some users, denying them service and making their coins unspendable. If Scrooge is greedy (as his novella namesake suggests), he could refuse to publish transactions unless they transfer some mandated transaction fee to him. Scrooge can also of course create as many new coins for himself as he wants. Or Scrooge could get bored of the whole system and stop updating the block chain completely.

The problem here is centralization. Although Scrooge is happy with this system, we, as users of it, might not be. While Scroogecoin may seem like an unrealistic proposal, much of the early research on cryptosystems assumed there would indeed be some central trusted authority, typically referred to as a *bank*. After all, most real-world currencies do have a trusted issuer (typically a government mint) responsible for creating currency and determining which notes are valid. However, cryptocurrencies with a central authority largely failed to take off in practice. There are many reasons for this, but in hindsight it appears that it's difficult to get people to accept a cryptocurrency with a centralized authority.

Therefore, the central technical challenge that we need to solve to improve on Scroogecoin and create a workable system is: Can we de-Scrooge-ify the system? That is, can we get rid of that centralized Scrooge figure? Can we have a cryptocurrency that operates like Scroogecoin in many ways but doesn't have any central trusted authority?

To do that, we need to figure out how all users can agree on a single published block chain as the authoritative history of all transactions. They must all agree on which transactions are valid, and which transactions have actually occurred. They also need to be able to assign IDs in a decentralized way. Finally, the minting of new coins also needs to be decentralized. If we can solve these problems, then we can build a currency that would be like Scroogecoin but without a centralized party. In fact, this would be a system much like Bitcoin.

FURTHER READING

Steven Levy's *Crypto* is an enjoyable nontechnical look at the development of modern cryptography and the people behind it:

Levy, Steven. *Crypto: How the Code Rebels Beat the Government—Saving Privacy in the Digital Age*. London: Penguin, 2001.

Modern cryptography is a rather theoretical field. Cryptographers use mathematics to define primitives, protocols, and their desired security properties in a formal way and to prove them secure based on widely accepted assumptions about the computational hardness of specific mathematical tasks. In this chapter we've used intuitive language to discuss hash functions and digital signatures. For the reader interested in exploring these and other cryptographic concepts in a more mathematical way and in greater detail, see:

Katz, Jonathan, and Yehuda Lindell. *Introduction to Modern Cryptography*, second edition. Boca Raton, FL: CRC Press, 2014.

For an introduction to applied cryptography, see:

Ferguson, Niels, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Hoboken, NJ: John Wiley & Sons, 2012.

Perusing the National Institute of Standards and Technology (NIST) standard that defines SHA-256 is a good way to develop an intuition for what cryptographic standards look like:

NIST. "Secure Hash Standards, Federal Information Processing Standards Publication." FIPS PUB 180-4. Information Technology Laboratory, NIST, Gaithersburg, MD, 2008.

Finally, here's the paper describing the standardized version of the ECDSA signature algorithm:

Johnson, Don, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)." *International Journal of Information Security* 1(1), 2001: 36–63.