© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

Contents

PREFACE vii

FOREWORD *The Long Road to Bitcoin* ix
JEREMY CLARK

CHAPTER 1 Introduction to Cryptography and Cryptocurrencies 1

CHAPTER 2 How Bitcoin Achieves Decentralization 27

CHAPTER 3 Mechanics of Bitcoin 51

CHAPTER 4 How to Store and Use Bitcoins 76

CHAPTER 5 Bitcoin Mining 104

CHAPTER 6 Bitcoin and Anonymity 138

CHAPTER 7 Community, Politics, and Regulation 168

CHAPTER 8 Alternative Mining Puzzles 190

CHAPTER 9 Bitcoin as a Platform 213

CHAPTER 10 Altcoins and the Cryptocurrency Ecosystem 242

CHAPTER 11 Decentralized Institutions: The Future of Bitcoin? 272

CONCLUSION 286

ACKNOWLEDGMENTS 287

ABOUT THE AUTHORS 289

INDEX 291

For general queries contact webmaster@press.princeton.edu.

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

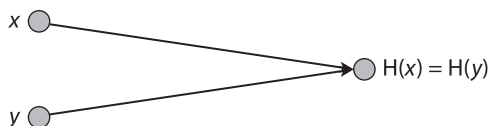

FIGURE 1.1. A hash collision. *x* and *y* are distinct values, yet when input into hash function *H*, they produce the same output.

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

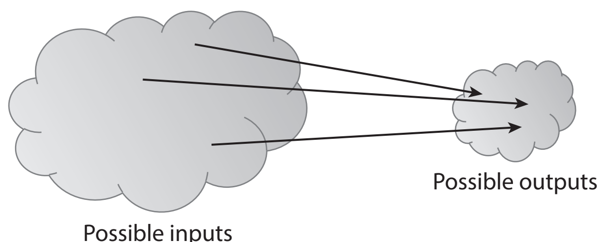

FIGURE 1.2. Inevitability of collisions. Because the number of inputs exceeds the number of outputs, we are guaranteed that there must be at least one output to which the hash function maps more than one input.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

(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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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,

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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:

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

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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

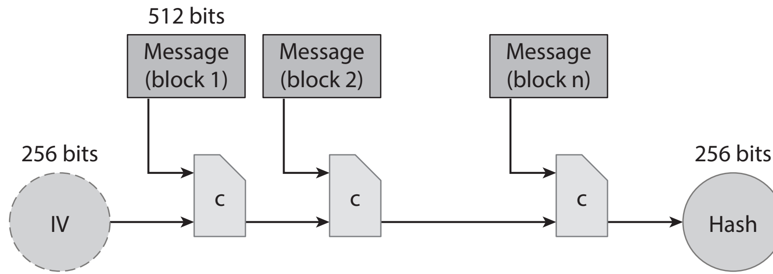

FIGURE 1.3. SHA-256 hash function (simplified). SHA-256 uses the Merkle-Damgård transform to turn a fixed-length collision-resistant compression function into a hash function that accepts arbitrary-length inputs. The input is padded, so that its length is a multiple of 512 bits. IV stands for initialization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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

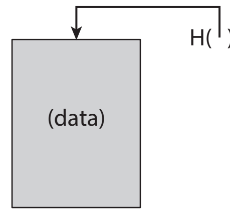

FIGURE 1.4. Hash pointer. A hash pointer is a pointer to where data is stored together with a cryptographic hash of the value of this data at some fixed point in time.

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.

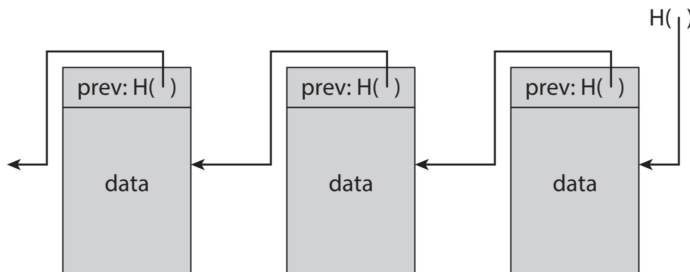

FIGURE 1.5. Block chain. A block chain is a linked list that is built with hash pointers instead of pointers.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

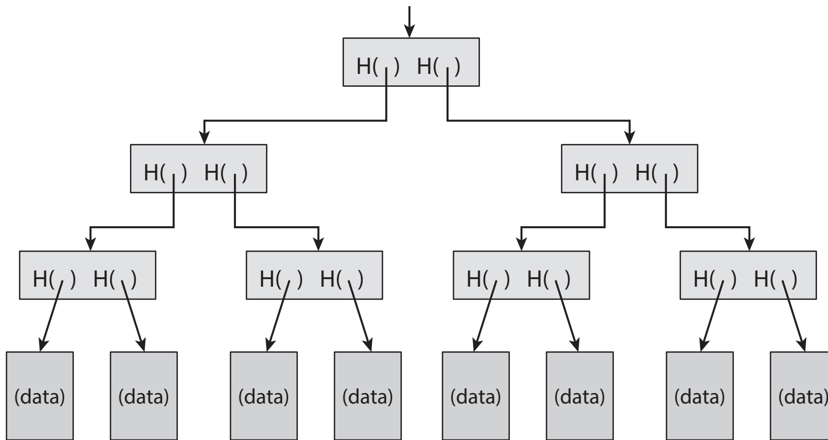

FIGURE 1.7. Merkle tree. In a Merkle tree, data blocks are grouped in pairs, and the hash of each of these blocks is stored in a parent node. The parent nodes are in turn grouped in pairs, and their hashes stored one level up the tree. This pattern continues up the tree until we reach the root node.

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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

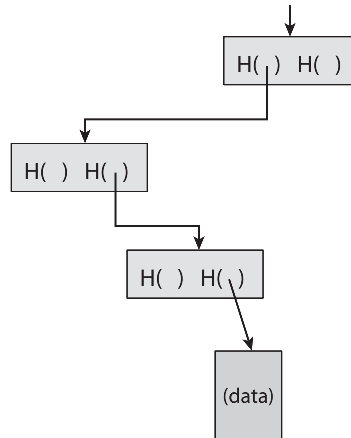

FIGURE 1.8. Proof of membership. To prove that a data block is included in the tree only requires showing the blocks in the path from that data block to the root.

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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

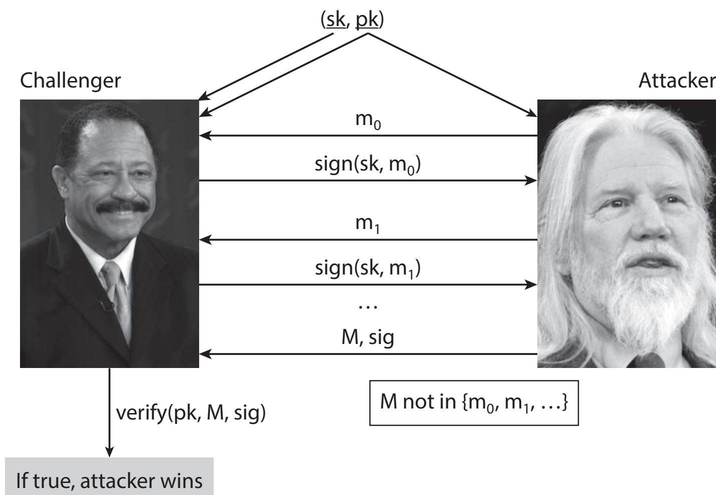

FIGURE 1.9. Unforgeability game. The attacker and the challenger play the unforgeability game. If the attacker is able to successfully output a signature on a message that he has not previously seen, he wins. If he is unable to do so, the challenger wins, and the digital signature scheme is unforgeable. 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.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

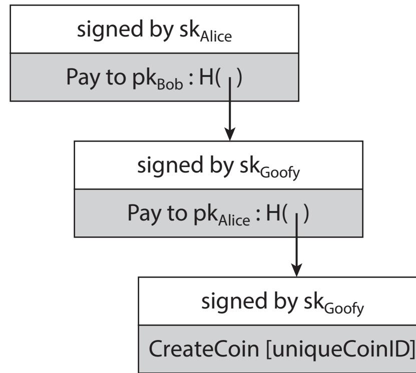

FIGURE 1.10. Goofycoin coin. Shown here is a coin that's been created (bottom) and spent twice (middle and top).

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).

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

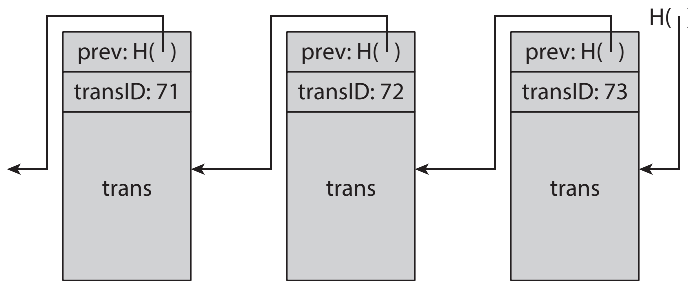

FIGURE 1.11. Scroogecoin block chain.

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).

transID: 73		type:CreateCoins	
coins created			
num	*value*	*recipient*	
0	3.2	0x…	← coinID 73(0)
1	1.4	0x…	← coinID 73(1)
2	7.1	0x…	← coinID 73(2)

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.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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-

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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.

For general queries, contact webmaster@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

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.

For general queries, contact webmaster@press.princeton.edu

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

Index

Italic page numbers refer to figures and tables.

4Chan, 139

Advanced Encryption Standard, 192
algorithms: altcoins and, 243, 266; anonymity and, 149; cryptography and, 3–4, 7, 15, 17–19, 26; decentralization and, 31–34, 38, 50, 273, 284; flooding, 67–69; mining and, 110, 194–95, 200–201, 208; networks and, 67, 69; platform issues and, 218–19; proof of work and, 243 (*see also* proof of work); protocol limitations and, 72–73; puzzles and, 200; storage and, 81; stylometry and, 176
altcoin infanticide, 244, 253, 256
altcoins: algorithms and, 243, 266; atomic cross-chain swaps and, 257–60; attacks and, 251, 253, 256–57, 269; Bitcoin and, 250–52, 260–63; block chains and, 242, 244, 246–47, 251, 253–55, 257–70; bootstrapping and, 244–45, 248–49, 253–54, 256, 260; cash and, 246; competition and, 251–52; consensus and, 242–43, 270–71; contesting a transfer and, 261–62; data structures and, 268–70; decentralization and, 42–43; deposits and, 258–59; double spending and, 244; escrow and, 247, 260; forks and, 171–73, 242–44, 248–49, 252–53, 256, 260, 262, 266, 270; hash functions and, 257, 270; hash pointers and, 254–55, 269; history of, 242–47; how to launch, 243–44; initial allocation and, 245–47; market capitalization and, 250; mathematics and, 267; merge mining and, 253–57; metadata and, 268; mining and, 242–57, 261–62, 266–70; nodes and, 242, 247, 260–62; nonces and, 257; payments and, 242, 244, 251, 261, 263, 267–69; prediction markets and, 263, 268; private keys and, 246–47, 250; profits and, 245, 248, 253, 256, 270; proof of work and, 243, 257, 260–63, 270; public keys and, 265; pump-and-dump scams and, 244–45; puzzles and, 248–56, 270; reasons for launching, 243; SHA-256 and, 250, 253, 256; sidechains and, 260–63, 270, 278; signatures and, 246, 258–59; smart contracts and, 263–70; switching costs and, 252; third parties and, 250–51; transaction fees and, 266; valid blocks and, 253; verification and, 260–62, 268; virtual machines and, 265–66, 270; wallets and, 247, 251–52; withdrawals and, 265–66. *See also specific coins*
Amazon, xi
AMD, 192
anonymity, xv; algorithms and, 149; attaching real-world identities and, 147–48; attacks and, 32–33, 40–41, 149, 154–55, 157, 164–65; banks and, 141–42, 152; block chains and, 139–52, 156–58, 161–64; bootstrapping and, 155; cash and, xiii–xiv, 142–43, 159–60, 163–66; chunk size and, 154, 157, 165; clusters and, 145–49, 159, 164; CoinJoin and, 145, 156–59, 165–66, 257; competition and, 142; consensus and, 159; crime and, 142–43, 178–81, 240; cryptocurrencies and, 138, 141–43, 159–60, 163, 165–67; deanonymization and, 140–51, 154, 219; decentralization and, 142–43; defining, 138–39; deposits and, 148, 151–53, 160; double spending and, 142, 157, 162, 164; ethics of, 138–42, 165; fiat currencies and, 142; forks and, 159; high-level flows and, 158–59; identity and, 139–41, 148–52; idioms of use and, 146–47; joint control and, 145, 279; legal issues and, 142, 149, 152; linking and, 144–46; mathematics and, 160, 165; merge avoidance and, 158–59; miners and, 142, 154, 159–60, 162–64; mixing and, 151–59; Mt. Gox and, 62, 90, 147–48; need for, 141–42; nodes and, 149–50; NSA and, 138; payments and, 140–42, 146–47, 158–59; peer-to-peer networks and, 149, 151, 155; privacy and, 138–44, 149–54, 159, 164, 166–67; private keys and, 144, 156; proof of work and, 157; pseudonyms and, 32, 46, 139–44, 152, 164–65, 176, 180, 280; public keys and, 139, 143–44, 163; puzzles and, 160; Satoshi and, xxii–xxvi; shared spending and, 145, 147; side channels and, 140, 153, 157–58, 164–65; signa-

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

anonymity (*cont.*)
tures and, 142, 156, 162; Silk Road and, 165, 179–81, 189; smart property and, 219–24; stealth addresses and, 144; Sybils and, 32–33, 40–41; tagging and, 148–49; taint analysis and, 141; Tor and, 143, 150, 153, 157, 167, 179–81; transaction fees and, 140, 154, 156, 164; transaction graph analysis and, 149, 151, 164–66, 219, 222, 269; unlinkability and, 81, 139–40, 144, 151, 157–59, 164; wallets and, 139, 141, 144–48, 151–55, 165; Wikileaks and, 138, 143–44; withdrawals and, 151–52; Zerocash and, 143, 159, 163–66, 282; Zerocoin and, 143, 159–66; zero-knowledge proofs and, 1, 160–64, 166, 229; zk-SNARKs and, 163–64
anonymity set, 140–41, 154–55
anti-money laundering (AML), 181–83
antitrust law, 186
AOL, 27
append-only log, 22–23, 51, 213–19, 247
application layers, 149–51
application-specific integrated circuits (ASICs): altcoins and, 248–49, 256; ASIC honeymoon and, 197; ASIC-resistant puzzles and, 190, 192–98, 208, 211, 249; mining and, 116–22, 190–98, 208, 211
asymmetric information, 184
atomic cross-chain swaps, 257–60
atomicity, 275–76, 279
attacks: 51 percent attacker and, 48–49, 128–30, 132, 197, 208–11; altcoins and, 251, 253, 256–57, 269; anonymity and, 32–33, 40–41, 149, 154–55, 157, 164–65; block-discarding, 204–5; checkpointing and, 210; clairvoyance and, 214–16; cryptography and, 1, 16–17, 22; decentralization and, 32–37, 41, 43, 48–49, 283; denial-of-service (DOS), 34, 157, 253; double spending and, 22 (*see also* double spending); exchange rate and, 132; fork, 131–36, 210 (*see also* forks); hackers and, 86, 90, 152, 165, 203, 218, 267, 275; illicit content and, 217–18; mining and, 127, 131–36, 191, 193, 195–98, 203–6, 209–10; networks and, 69; phishing, 283; platform issues and, 214, 216, 233–34; practical countermeasures and, 132; profits and, 233; protocol limitations and, 73; sabotage, 205–6; stake-grinding, 209–10; storage and, 82; Sybil, 32–33, 40–41; temporary block-withholding, 133–34; vigilante, 205
automobiles, 273, 273–74

Back, Adam, xix
bank runs, 89–90
bankruptcy, 90, 175
banks: anonymity and, 141–42, 152; blocks and, 61, 66; central, 1, 25; double-spending and, 62; exchanges and, 88–91, 99, 102; government-issued ID and, 99; green addresses and, 61; payment services and, 96; platform issues and, 220–21; regulation and, 90–91, 99, 168, 175, 178; state determination and, 169; traditional, 90–91, 141, 152, 269; trust and, 25
bartering, ix
base-58 notation, 77, 83
Basecoin, 159–64, 260, 282
beacons, 229–34, 268
Bernoulli trials, 43
Betamax, 252
binding, 6–8, 280
Bitcoin: altcoins and, 250–52, 260–63; as append-only log, 22, 51, 213–19, 247; beacons and, 229–34; colored coins and, 221–24, 277; consensus and, 168–70 (*see also* consensus); contesting a transfer and, 261–62; CreateMarket and, 236–38; crime and, 142–43, 178–81, 240; cypherpunks and, xvi–xvii, xxiv, 175–76, 188, 282; data feeds and, 234–39; deanonymization of, 143–51; decentralization and, 27–28, 272–85 (*see also* decentralization); denominations of, 46; escrow and, 60–64; forks and, 69, 73–75 (*see also* forks); future issues and, 272–85; governments' notice of, 178–81; growth of, 176–78; illicit content and, 217–18; integration routes for, 275–78; licenses and, 186–89; lotteries and, 63, 224–34, 241; mandatory reporting and, 182–83; mining and, 190 (*see also* mining); OpenAssets and, 221–23, 241; order books and, 231, 236, 240–41, 268; overlay currencies and, 218–19, 247; platform issues and, 213–41 (*see also* platform issues); power of, 286; prediction markets and, 234–41; roots of, 175–78; Satoshi Nakamoto and, xvi–xvii, xxii–xxvi, 18, 46, 119, 176; sidechains and, 260–63, 270, 278; as smart property, 219–24; stakeholders and, 138, 173–75, 186, 203, 208, 244; switching costs and, 252; trust and, 280; zero-knowledge proofs and, 1, 160–64, 166, 229
"Bitcoin: A Peer to Peer Electronic Cash System" (Nakamoto), 176
Bitcoin Core, 72, 145, 170–71, 174–75, 203, 210
Bitcoin Foundation, 174–75
Bitcoin Improvement Proposals (BIPs), 170, 174
Bitcoin mechanics: block chains and, 53, 56, 59–75, 286; block rewards and, 39–40, 45–46, 49, 66, 77, 98, 105, 127–28, 136, 205, 234; block-size conundrum of, 75; bootstrapping and, 59; capital

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

controls and, 178; change address and, 52–53, 62, 145–47, 268; consensus and, 51, 64, 68, 75; consolidating funds and, 53; data structures and, 51–53, 64, 66, 71; green addresses and, 61–63; hash functions and, 56–57, 73; hash pointers and, 52, 54, 64–66; improvements and, 72–75; joint payments and, 53; latency and, 30–31, 36, 42–43, 46, 68–69, 132, 150, 213; mathematics and, 86; miners and, 51–56, 60–65, 68–74, 97–98; networks and, 66–72; nodes and, 59, 66–75; nonces and, 65; parameterizable cost and, 42–45; Pay-to-Script-Hash and, 59–60, 74, 218, 221; peer-to-peer networks and, 29, 59, 66–67; protocol limitations and, 72–75; puzzles and, 64; SHA-256 and, 57, 73; traditional assumptions and, 31–32; transactions and, 51–54; valid blocks and, 68, 73–74; verification and, 53, 56, 58, 71

Bitgold, xxii–xxiv

BitTorrent, xi

Black Hat conferences, 149–50

blacklisting, 135–36

blockchain.info, 88

block chains: 51 percent attacks and, 48–49, 128–30, 132, 197, 208–11; altcoins and, 242, 244, 246–47, 251, 253–55, 257–70; alternative, 277–78; anonymity and, 139–52, 156–58, 161–64; append-only ledger and, 22–23, 51, 213–19, 247; application layers and, 149–51; banks and, 61, 66; Bitcoin mechanics and, 53, 56, 59–75, 286; bootstrapping and, 47–48; certificates and, xx–xxi, 280; coinbase transactions and, 65–66, 74, 88, 94, 105–7, 125, 204–6, 219, 254–56; CoinJoin and, 156–58; community and, 168–73, 181; competition and, 106; consensus and, 32–38, 104; contesting a transfer and, 261–62; cryptography and, 11–13, 17, 22–25; deanonymization and, 149–51; decentralization and, 30, 32–38, 46–50, 272–78, 281–85; efficient verification and, 53; exchanges and, 88–89; genesis block and, 12, 77, 171–72, 201, 210, 219, 242; hard forks and, 47, 73–75, 135, 172–73, 241, 252, 266, 270; hash pointers and, 11–12; illicit content and, 217–18; integration routes for, 275–78; maintaining, 104; Merkle trees and, 12–14, 64–65, 92–93, 105–7, 201–2, 204, 217, 255, 269; mining and, xxii, 42, 104–5, 108, 130, 131–32, 133–34, 135, 191, 200, 207, 210; nodes and, 43 (*see also* nodes); orphan block and, 36, 46, 134; overlay currencies and, 218–19, 247; parameterizable cost and, 42–45; platform issues and, 217–19, 223–24, 232; Poisson distribution and, 43–44, 124; politics and, 66, 75; proof of membership and, 13–14; proof of nonmembership and, 14–15; Satoshi and, xxiii–xxiv; signatures and, 205–6; smart contracts and, 263–70; soft forks and, 47, 73–74, 159, 172–73, 241, 256, 260; storage and, 76, 79, 81–82, 86; tamper-evident logs and, 11–12, 83; transaction fees and, 65, *66*, 97–98, 105

block-discarding attacks, 204–5

block reward, 39–40, 45–46, 49, 66, 77, 98, 105, 128, 136, 205, 234

block size, 10, 70, 75, 243

Blu-ray, 251

b-money, xxii–xxiv

BOINC (Berkeley Open Infrastructure for Network Computing), 198

Bonneau, Joseph, 155

bootstrapping: altcoins and, 244–45, 248–49, 253–54, 256, 260; anonymity and, 155; Bitcoin mechanics and, 59; block chains and, 47–48; cryptocurrencies and, 47–48; decentralization and, 47–48, 59, 155, 197, 244–45, 248–49, 253–54, 256, 260; mining and, 197; networks and, 47–48

brain wallet, 81–83, 87

Brands, Stefan, xvi

bribery, 133, 279

Byzantine Generals Problem, 31

Café, xviii

Camenisch, Jan, xvi

capital controls, 178

cash, x–xi; advantages of, xiii–xiv; altcoins and, 246; anonymity and, xiii–xiv, 142–43, 159–60, 163–66; Chaum and, xiv–xv; community and, 169, 175–76, 178, 189; cryptography and, xiv–xv (*see also* cryptography); decentralization and, 28, 38, 272, 282, 284; exchanges and, 75, 99; mining and, 123, 133

certificates, xx–xxi, 280

change address, 52–53, 62, 145–47, 268

Chaum, David, xiv–xvi, xxv, 142–43, 175

checkpointing, 210

chess, 267–68

Chrome, 248

chunk size, 154, 157, 165

ciphers, 84, 192, 264

clairvoyance, 214–16

Clark, Jeremy, ix–xxvii

clusters, 145–49, 159, 164

CoiledCoin, 253, 256

coin-age, 208–10

coinbase transactions, 65–66, 74, 88, 94, 105–7, 125, 204–6, 219, 254–56

Coin Center, 175

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

CoinJoin, 145, 156–59, 165–66, 257
collision resistance, 2–5
colored coins, 221–24, 277
CommitCoin, 216–17
commitments, 6–8, 19, 161–64, 214, 216–17, 222, 225–26, 234, 258–59
community: block chains and, 168–73, 181; cash and, 169, 175–76, 178, 189; competition and, 173, 186; consensus and, 168–70, 173–75; cryptocurrencies and, 168–69, 172, 174; deposits and, 181; escrow and, 180–81; miners and, 172–73, 188; payments and, 174, 178–80; privacy and, 175, 189; public keys and, 175; Satoshi Nakamoto and, 171; trust and, 280; valid blocks and, 168
compatibility, 159
competition: altcoins and, 251–52; anonymity and, 142; blocks and, 105; community and, 173, 186; decentralization and, 27, 41–43, 47, 278–79, 281, 285; hash puzzles and, 41; mining and, 105, 110, 117, 127, 133, 196, 212; supply/demand issues and, 101–2
compression function, 9–10, 18, 111
CompuServe, 27
consensus, 99; altcoins and, 242–43, 270–71; anonymity and, 159; Bitcoin mechanics and, 51, 64, 68, 75; breaking traditional assumptions and, 31–32; Byzantine Generals Problem and, 31; community and, 168–70, 173–75; decentralization and, 28–40, 46–50, 242, 275, 277, 282; distributed, 28–32, 38, 47, 242; fiat currencies and, 168–71; history and, 168; without identity, 32–38; implicit, 33–38; latency and, 30–31, 36, 42–43, 46, 68–69, 132, 150, 213; mining and, 104–5, 108, 123, 131–32, 135, 190, 195, 198, 200–204, 206, 210; nodes and, 28–40, 46–50, 168; platform issues and, 218–19; public keys and, 29; rules and, 168; Sybil attacks and, 32–33, 40–41; theft and, 34; Tinkerbell effect and, 169, 244; value of coins and, 168
consolidating funds, 53
counterfeiting, 1, 220
Coursera, 286
CPU mining, 107, 111–12, 118, 248
cracking, 82, 103, 264
CreateCoins, 21–24, 39, 52, 65
CreateMarket, 236–38
credit cards, xi–xiii, 72, 139, 285
crime, 240; anonymity and, 142–43; anti-money laundering (AML) and, 181–83; Silk Road and, 165, 179–81, 189
crowd-funding services, 264, 275–76, 284
cryptocurrencies, 286; altcoins and, 75, 242–70 (*see also* altcoins); anonymity and, 138, 141–43, 159–60, 163, 165–67; Bitcoin–altcoin interactions and, 251–52; bootstrapping and, 47–48; community and, 168–69, 172, 174; contesting a transfer and, 261–62; crime and, 142–43, 178–81, 240; cryptography and, 1–3, 10, 15, 18–25; decentralization and, 27, 41, 43, 47–48, 278; ecosystem of, 242–70; mining and, 117, 137, 193–201, 206–11; nodes and, 217, 219; platform issues and, 234, 238–39, 241; politics and, 198; proof of stake and, 41, 206–11; Satoshi Nakamoto and, xvi–xvii, xxii–xxvi, 18, 46; security and, 1–3, 10, 15, 18–25 (*see also* security); sidechains and, 260–63, 270, 278; storage and, 76, 79–80, 83–85, 198; simple, 20–25; virtual machines and, 265–66, 270
cryptography: Advanced Encryption Standard and, 192; algorithms and, 3–4, 7, 15, 17–19, 26; attacks and, 1, 16–17, 22; automobiles and, 273; base-58 notation and, 77, 83; beacons and, 229–34, 268; binding and, 6–8, 280; block chains and, 11–13, 17, 22–25; Chaum and, xiv–xvi, xxv, 142–43, 175; ciphers and, 84, 192, 264; collision resistance and, 2–5; commitments and, 6–8, 19, 161–64, 214, 216–17, 222, 225–26, 234, 258–59; compression function and, 9–10, 18, 111; cracking and, 82, 103, 264; cryptocurrencies and, 1–3, 10, 15, 18–25; data structures and, 10–15, 21–22; double spending and, 22–25; Elliptic Curve Digital Signature Algorithm (ECDSA) and, 17–19, 26, 73, 80, 144, 216, 273, 276; encryption and, xi–xiii, xvii, 19, 84, 88, 179, 192; Fiat and, xv–xvi; genesis block and, 12, 77, 171–72, 201, 210, 219, 242; guarantees and, 159; hackers and, 86, 90, 152, 165, 203, 218, 267, 275; hash functions and, 2–26; hiding and, 2, 5–8, 19, 130; HTTP and, xii–xiii; identity and, 19–20; initialization vector (IV) and, 9, *10*; lotteries and, 229; mathematics and, 1–2, 8, 26; Merkle-Damgård transform and, 9–10, 12; Merkle trees and, xvi, 12–14, 64–65, 92–93, 105–7, 201–2, 204, 217, 255, 269; message digests and, 4–5, 17; Naor and, xv–xvi; nonces and, 6–8; politics and, 285; prime numbers and, 84–85, 163, *199*, 200–201; privacy and, 20; private keys and, 18 (*see also* private keys); proof of membership and, 13–14; proof of nonmembership and, 14–15; public keys and, 15–24, 29 (*see also* public keys); puzzles and, 2, 8–10, 41, 198; QR codes and, 77–78; random oracle model and, 10; RSA, xx, 163; secret sharing and, 83–87; SHA-256 and, 9–10; signatures and, 1, 15–26, 34, 80, 220, 229, 273; storage and, 76, 79–80, 83–85, 198;

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

tampering and, 1, 5, 11–13, 83, 213, 230, 247; threshold, 86–87; unforgeability and, 15–17; verification and, 14–18; zero-knowledge proofs and, xvi, 1, 160–64, 166, 229; zk-SNARKs and, 163–64
CryptoNote, 144
Cuckoo Cycle, 195, 211
Cunningham chain, 200–201
cyberbucks, xvi
CyberCash, xii–xiii, xvi
cypherpunks, xvi–xvii, xxiv, 175–76, 188, 282

Dai, Wei, xxii, xxiv
Dark Wallet, 144
data feeds, 234–39
data structures: altcoins and, 268–70; Bitcoin mechanics and, 51–53, 64, 66, 71; cryptography and, 10–15, 21–22; decentralization and, 34; distributed problem and, 169; Ethereum and, 269; genesis block and, 12, 77, 171–72, 201, 210, 219, 242; hash pointers and, 10–15; Merkle trees and, 12–14, 64–65, 92–93, 105–7, 201–2, 204, 217, 255, 269; mining and, 195; platform issues and, 213; proof of membership and, 13–14
deanonymization: anonymity and, 140–41, 143–51, 154, 219; attaching real-world identities and, 147–48; Bitcoin and, 143–51; block chains and, 149–51; clusters and, 145–49, 159, 164; identifying individuals and, 149; idioms of use and, 146–47; joint control and, 145, 279; linking and, 144–46; network-layer, 149–51; shared spending and, 145, 147; side channels and, 140, 153, 157–58, 164–65; stealth addresses and, 144; tagging and, 148–49; transaction graph analysis and, 149, 151, 164–66, 219, 222, 269
decentralization: algorithms and, 31–34, 38, 50, 273, 284; anonymity and, 142–43; atomicity and, 275–76, 279; attacks and, 32–37, 40–41, 43, 48–49, 128–30, 197, 208–11, 283; benefits of, 282–85; block chains and, 30–38, 46–50, 272–78, 281–85; bootstrapping and, 47–48, 59, 155, 197, 244–45, 248–49, 253–54, 256, 260; breaking traditional assumptions and, 31–32; Byzantine Generals Problem and, 31; cash and, 28, 38, 272, 282, 284; centralization and, 27–28; competition and, 27, 41–43, 47, 278–79, 281, 285; consensus and, 28–40, 46–50, 242, 275, 277, 282; cost of mining and, 45–47; crowd-funding services and, 264, 275–76, 284; cryptocurrencies and, 27, 41, 43, 47–48, 278; data structures and, 34; deposits and, 258–59; disintermediation and, 275, 278–79, 281; dispute mediation and, 278–79; double spending and, 34–38, 46, 49; fiat currencies and, 47; forks and, 47–48, 277; future institutions and, 272–85; hash functions and, 35, 41–43, 276–77; high-level flows and, 158–59; identity and, 19–20, 32–38, 41; incentives and, 38–45; legal issues and, 240, 279, 282, 284–85; levels of, 278; lotteries and, 33; mathematics and, 43; miners and, 277; mixing and, 155–59; nodes and, 28–49; nonces and, 41–44; order books and, 231, 236, 240–41, 268; parameterizable cost and, 42–45; paying for a proof and, 276–77; payments and, 34–37, 39, 48, 274, 276–77, 281–82; peer-to-peer networks and, 28–32, 36, 42, 46–50; politics and, 282, 285; prediction markets and, 236–37, 279–82; privacy and, 284; private keys and, 273, 276, 283; proof of work and, 38–45, 50; public keys and, 29, 34, 273, 276; puzzles and, 41–43, 46–47, 50; security and, 279–80, 283–84; signatures and, 34, 48, 273–74, 276, 279; smart property and, 273–74, 281–85; StorJ and, 282; template for, 278–82; third parties and, 274; transaction fees and, 39–40, 45–46, 277; trust and, 280; valid blocks and, 30, 39, 48; wallets and, 28; Zerocoin and, 281–82
denial-of-service (DOS) attacks, 34, 157, 253
deposits: altcoins and, 258–59; anonymity and, 148, 151–53, 160; community and, 181; decentralization and, 258–59; exchanges and, 88–93, 100; mining and, 209; payment services and, 96–97; platform issues and, 226, 234
DigiCash, xvi–xviii
Digigold, xix
disintermediation, 275, 278–79, 281
disputes, 60–61, 214, 238, 274, 278–80, 283–85
distributed consensus, 28–32, 38, 47, 242
Dogecoin, 249–50
domain names, 29, 223–24, 248, 257
double spending, xiv–xvi; altcoins and, 244; anonymity and, 142, 157, 162, 164; append-only ledger and, 22–23, 51, 213–19, 247; cryptography and, 22–25; decentralization and, 34–38, 46, 49; mining and, 131–33; networks and, 68–69; platform issues and, 218; scripts and, 62–63
drugs, 165, 179–81, 189
DSA algorithm, 17–18
Dwork, Cynthia, xix

ecash, xvi–xviii, xxv, 142–43
economic issues, vii; asymmetric information and, 184; Bitcoin–altcoin interactions and, 251–52; credit cards online and, xi–xiii; crowd-funding services and, 264, 275–76, 284; decentralization and, 45 (*see also* decentralization); exchanges

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

economic issues (*cont.*)
and, 99; fungible goods and, 219; investors and, 72, 102, 173–74, 244–45; long-term changes and, 203; mining and, 45, 117–18, 123, 257; minting money out of air and, xviii–xx; Pareto improvement and, 183; prediction markets and, 235; proof of work and, 203; stakeholders and, 138, 173–75, 186, 203, 208, 244; switching costs and, 252; traditional financial arrangements and, ix–xi
Edison, Thomas, 252
efficiency, 184
e-Gold, xviii–xix, xxv–xxvi
electricity, 45, 47, 115, 117–24, 128, 130, 192, 203, 207, 211
Eligius, *129*, 253, 256
Elliptic Curve Digital Signature Algorithm (ECDSA), 17–19, 26, 73, 80, 144, 216, 273, 276
encryption, xi–xiii, xvii, 19, 84, 88, 179, 192
energy: bottom-up approach and, 121–22, 198, 203; cooling equipment and, 120–21; ecological issues and, 119–23; electric, 45, 47, 114, 117–24, 128, 130, 192, 203, 207, 211; embodied, 120; estimating usage of, 121–22; joule measurement of, 119, 121; Landauer's principle and, 119–20; repurposing, 123; Three Gorges Dam and, 122; top-down approach and, 121; waste and, 122; wattage and, 121, 198, 203
entropy, 6, 8–9, 82, 214, 232
equiprobable solution space, 199
escrow: altcoins and, 247, 260; Bitcoin mechanics and, 60–64; community and, 180–81; platform issues and, 227; scripts and, 60–61
Ethereum, 210, 278; chess in, 267–68; data structures and, 269; Frontier project and, 269–70; loop support and, 266; Namecoin and, 263, 265; Patricia tree and, 269; security and, 266–67; smart contracts and, 263–70; state and account balances in, 268–69; virtual machines and, 265–66, 270
ethics, 138–42, 165
exchanges: banks and, 88–91, 99, 102; block chains and, 88–89; cash and, 75, 99; currency markets and, 99–102; deposits and, 88–93, 100; fiat currencies and, 89, 99–102, 178; fractional reserve and, 88–89, 91; hash pointers and, 92–93; Mt. Gox and, 62, 90, 147–48; nodes and, 93; Ponzi schemes and, 89–90; privacy and, 91–94; private keys and, 91; proof of liabilities and, 91–94; proof of reserve and, 91, 93–94; security and, 274–75; Silk Road and, 180; simple market behavior and, 101–2; storage and, 87–94, 99–102; supply and demand issues and, 99–101; wallets and, 87–94; withdrawals and, 88–90

Facebook, 27, 29
FBI, 180–81
feather forking, 135–36
Fiat, Amos, xv–xvi
fiat currencies: anonymity and, 142; central banks and, 1, 25; consensus and, 168–71; decentralization and, 47; exchanges and, 89, 99–102, 178; miners and, 245; payment services and, 94–97; politics and, 183, 188; pre-sales and, 245; regulation and, 183; transfers and, 88
field-programmable gate arrays (FPGAs), 114–16, 118, 192, 197
financial data beacons, 231
Firefox, 248
FirstVirtual, xii, xvi
"Fistful of Bitcoins, A: Characterizing Payments among Men with No Names" (Meiklejohn et al.), 147–48, 166
flooding algorithm, 67–69
forgery, 15–18, 25, 34, 67, 240–41
forks: altcoins and, 242–44, 248–49, 252–53, 256, 260, 262, 266, 270; anonymity and, 159; Bitcoin mechanics and, 69, 73–75; checkpointing and, 210; decentralization and, 47–48, 277; feather, 135–36; hard, 47, 73–75, 135, 172–73, 241, 252, 266, 270; mining and, 131–36, 195, 209–10; open-source software and, 171–73; overlay currencies and, 277; platform issues and, 233, 241; soft, 47, 73–74, 159, 172–73, 241, 256, 260; software rules and, 171–73
fractional reserve, 88–89, 91
fraud, 91, 116, 245
fungibility, 219

gamers, 113
generateKeys, 15–16, 19, 80–81
genesis block, 12, 77, 171–72, 201, 210, 219, 242
GHash.IO, 128–30
GHOST protocol, 270
global time, 31
Goofycoin, 21–24
gossip protocol, 67
Götze, Mario, 215
GPU mining, 112–14, 192, 196, 248
green addresses, xiv, 61–63
Guy Fawkes signature scheme, 214

Haber, S., xx
hackers, 86, 90, 149–50, 152, 165, 203, 218, 267, 275
hard forks, 47, 73–75, 135, 172–73, 241, 252, 266, 270

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

Hashcash, xix–xx, xxiv
hash functions: altcoins and, 257, 270; binding and, 6–8, 280; Bitcoin mechanics and, 56–57, 73; collision resistance and, 2–5; commitments and, 6–8, 19, 161–64, 214, 216–17, 222, 225–26, 234, 258–59; compression, 9–10, 18, 111; cryptography and, 2–10, 12, 15–20, 26; decentralization and, 41–43, 276–77; hiding and, 2, 5–8, 19, 130; initialization vector (IV) and, 9, *10*; Merkle-Damgård transform and, 9–10, 12; message digests and, 4–5, 17; message size and, 17; mining and, 110–15, 120–22, 191–202, 208, 212, 250, 253, 256; modeling, 10; platform issues and, 213–14, 217, 232; properties of, 2–10; puzzle friendliness and, 8–10, 41, 198; random oracle model and, 10; search puzzles and, 8–9; SHA-256, 9–10, 57, 73, 82, 110–16, 120, 122, 191–202, 217, 250, 253, 256; storage and, 78–79, 82; targets and, 8, 41–45, 105–6, 125, 160, 191, 196, 202–6, 254–55, 262–63, 270; timestamping and, 213–14
hash pointers: altcoins and, 254–55, 269; Bitcoin mechanics and, 52, 54, 64–66; block chains and, 11–12; cryptography and, 10–15, 17, 21–23; data structures and, 10–15; decentralization and, 35, 41; exchanges and, 92–93; genesis block and, 12, 77, 171–72, 201, 210, 219, 242; Goofycoin and, 21–24; merge mining and, 255; Merkle trees and, 12–14; platform issues and, 213; proof of membership and, 13–14; proof of nonmembership and, 14–15; tamper-evident logs and, 11–12, 83
hash puzzles, 41–47, 50, 160, 232. *See also* mining
hash rate, 45, 47, 108–9, 116, 121–22, *125*, 244, 250–51
HD DVD, 251–52
Hearn, Mike, 158
hiding, 2, 5–8, 19, 130
high-level flows, 158–59
Hohenberger, Susan, xvi
HTML, 94–96
HTTP, xii–xiii
hype, 244–45, 286

IBM, xii
identity: anonymity and, 139–41, 148–52; consensus without, 32–38, 169; cryptography and, 19–20; decentralization and, 20, 32–38, 41; merchant ID and, 96; platform issues and, 216; real-world, 19–20, 29, 139–41, 149, 151, 182; Satoshi and, 176; Silk Road and, 180; storage and, 76; tax evasion and, 179; Ulbricht and, 180
idioms of use, 146–47
illicit content, 217–18
implicit consensus, 33–38
incentives: block rewards and, 39–40, 45–46, 49, 66, 77, 98, 105, 127, 136, 205, 234; miners and, 42–48; parameterizable cost and, 42–45; proof of work and, 38–45; transaction fees and, 39–40 (*see also* transaction fees)
inexhaustible puzzle space, 199–200
inflation, xix, 243
initialization vector (IV), 9, *10*
Instawallet, 62
Intel, 192
investors, 72, 102, 173–74, 244–45
IP addresses, 29, 32, 70, 143, 149–51, 223–24, 248

joint control, 145, 279
joint payments, 53
joules, 119, 121

Kaminsky, Dan, 149–50
Karma, *x*, xi
Keccak, 196
key stretching, 82
Know Your Customer (KYC), 182

Landauer's principle, 119–20
latency, 30–31, 36, 42–43, 46, 68–69, 132, 150, 213
laundering, xxvi, 130, 142, 166, 181–83
laundry, 152
Laurie, Ben, xvii
law enforcement, 1, 135, 143, 149, 168, 178–81, 283
ledgers, xx–xxiii; altcoins and, 268–69; anonymity and, 141, 164; append-only, 22–23, 51, 213–19, 247; Bitcoin mechanics and, 51–53; cryptography and, 22, 24; decentralization and, 27–28, 30, 32, 47, 49
legal issues: anonymity and, 142, 149, 152; antitrust and, 186; centralized order books and, 240; company stock and, 223; competition and, 186; decentralization and, 240, 279, 282, 284–85; drugs and, 179, 181; illicit content and, 217–18; law enforcement and, 1, 135, 143, 149, 168, 178–81, 283; mining and, 135, 204; money laundering and, xxvi, 142, 166, 181–83; physical property and, 223; pornography and, 217–18; regulation and, 179, 181, 183, 186 (*see also* regulation); selling votes and, 204; Silk Road and, 165, 179–81, 189
lemons market, 184–86
lender of last resort, 90

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

libertarianism, 175, 188
Liberty Reserve, xxv–xxvi
licenses, 170, 186–89
LinkedIn, 27
linking, 144–46
Litecoin, 119, 193, 196, 248–49, 252, 256
lock time, 63–64
lotteries: beacons and, 229–34; Bitcoin and, 63, 224–34, 241; cryptographic beacons and, 229; decentralization and, 33; fairness and, 225–27; financial data and, 231–32; military draft, 227–29; natural phenomena and, 230–31; NBA draft, 227; NIST beacon and, 229–30; online coin flipping and, 225; secure multiparty, 63, 224–34, 241; secure multiparty computation and, 224–34, 241
Lucre, xvii
Madoff, Bernie, 90

MagicMoney, xvii
MasterCard, xviii
mathematics: algorithms and, 3–4 (*see also* algorithms); altcoins and, 267; anonymity and, 160, 165; Bernoulli trials and, 43; Bitcoin mechanics and, 86; Cunningham chain and, 200–201; cryptography and, 1–2, 8, 26; decentralization and, 43; mining and, 191, 195, 201; Poisson distribution and, 43–44, 124, *125*; prime numbers and, 84–85, 163, *199*, 200–01
Maxwell, Greg, 282
McCain, John, 236
memory-bound puzzles, 193, 195, 211
memory-hard puzzles, 193–96, 211, 248, 270
memoryless process, 191
merge avoidance, 158–59
merge mining, 246, 248, 253–57, 267, 270
Merkle, Ralph, 12
Merkle-Damgård transform, 9–10, 12
Merkle trees: cryptography and, xvi, 12–14, 64–65, 92–93, 105–7, 201–2, 204, 217, 255, 269; Patricia, 269; proof of membership and, 13–14; proof of nonmembership and, 14–15; sorted, 14
message digests, 4–5, 17
metadata: altcoins and, 268; platform issues and, 220–22; protocol limitations and, 74; transactions and, 53–54, 64
micropayments, xiv, 62–64, 268
Microsoft, xii
military draft lottery, 227–29
min-entropy, 6, 8–9, 214
miners: altcoins and, 244–57, 261–62, 266–69; anonymity and, 142, 154, 159–60, 162–64; behavioral models of, 43; Bitcoin mechanics and, 51–56, 60–65, 68–74, 97–98; block chain maintenance and, 104–5; candidate block assemblage and, 105; community and, 172–73, 188; decentralization and, 277; fiat currencies and, 245; gamers and, 113; incentives and, 42–48; listening for transactions and, 104; Nash equilibrium and, 43; platform issues and, 216–19, 222–23, 232–34, 238, 240; profit and, 105; as stakeholders, 173; task of, 104–19
mining: 51 percent, 48–49, 128–30, 131–32, 197, 208–11; algorithms and, 110, 194–95, 200–201, 208; altcoins and, 242–57, 262, 267, 270; application-specific integrated circuits (ASICs) and, 116–22, 190–98, 208, 211, 248–49, 256; attacks and, 127, 131–36, 191, 193, 195–98, 203–6, 209–10; blacklisting and, 135–36; block chains and, 104–5, 108, 130, 131–32, 133–34, 135–36, 191, 200, 207, 210; block-discarding attacks and, 204–5; bootstrapping and, 197; bottom-up approach and, 121–22, 198, 203; cash and, 123, 133; competition and, 105, 110, 117, 127, 133, 196, 212; consensus and, 104–5, 108, 123, 131–32, 135, 190, 195, 198, 200–204, 206, 210; cost of, 28, 42, 45–47, 123, 195; CPU, 107, 111–12, 118, 248; cryptocurrencies and, 117, 137, 193–201, 206–11; Cunningham chain and, 200–201; data structures and, 195; deposits and, 209; difficulties of, 107–10; double spending and, 131–33; ecological issues and, 119–23; economic issues and, 45; energy consumption and, 119–24; equiprobable solution space and, 199; essential puzzle requirements and, 190–92; field-programmable gate arrays (FPGAs) and, 114–16, 118, 192, 197; forks and, 131–36, 195, 209–10; future issues and, 118–19; gold, 118, 119; GPU, 112–14, 192, 196, 248; hardware for, 110–19; hash functions and, 110–15, 120–22, 191–202, 208, 212, 250, 253, 256; high variance and, 124, *125*; hopping and, 127–28; incentives for, 130–36; inexhaustible space and, 199–200; Landauer's principle and, 119–20; legal issues and, 135, 204; mathematics and, 191, 195, 201; memoryless process and, 191; merge, 246, 248, 253–57, 267, 270; modern professional, 117–19; negative externalities and, 198; nodes and, 104, 111, 113, 117, 125, 130, 134, 190, 203, 210; nonces and, 104–7, 111–13, 124, 199, 202; nonoutsourceable puzzles and, 203–6; nothing-at-stake problem and, 209–10; open problems and, 136; overclocking and, 113, 115; payments and, 126–27, 131–32, 206–7; peer-to-peer networks and, 117, 128; Poisson distribution and, 124, *125*; pools and, 107, 124–30,

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

203–6, 233, 253, 256–57, 262; power of, 250–51; pre-mining and, 244–45; private keys and, 205–6, 210; profits from, 45, 47, 105–6, 110, 112, 116–19, 124–25, 131, 133, 136, 190, 197, 205; progress free puzzles and, 191, 199, 201; proof of retrievability and, 201; proof of stake and, 206–11; proof of work and, 40–42, 193, 195, 198–203, 208, 211; proportional model and, 127; pseudocode for, *112*, *194*; public good and, 203; public keys and, 107, 202, 204–6; puzzles and, 64, 107, 119, 122, 190–211, 248–56, 270; sabotage and, 205–6; Satoshi Nakamoto and, 48, 204; at scale, 120–21; selfish, 134; SHA-256 and, 110–13, 116, 119, 120, 122, 191–202, 208, 250, 253, 256; shares and, 125–28; signatures and, 104, 205–6, 210; stake-grinding attacks and, 209–10; strategies for, 130–36; targets and, 105–6, 125, *126*, 191, 196, 202–6, 254; time-memory trade-offs and, 194–95; top-down approach and, 121; transaction fees and, 54, 97–98, 136, 203, 211; valid blocks and, 73–74, 105–6, 111–12, 113, 125–27, 133–34, 199, 204–5, 208, 210; verification and, 191, 195–96, 203; vigilante attacks and, 205; virtual, 206–11; waste and, 122
minting, 25, 65, 160–61
MIT license, 170
mixing: anonymity and, 151–59; automated client side and, 154; chunk size and, 154, 157, 165; CoinJoin and, 145, 156–59, 165–66, 257; decentralization and, 155–59; dedicated services for, 152–53; fees and, 154–55; guidelines for, 153–55; high-level flows and, 158–59; laundry and, 152; merge avoidance and, 158–59; online wallets as, 151–52; in practice, 155; series of, 153; Tor and, 153; tumblers and, 152
mix net, 150, 157
MojoNation, xi
Mondex, xviii
money laundering, xxvi, 142, 166, 181–83
Mt. Gox, 62, 90, 147–48
MULTISIG, 56–63, 74
multisignatures, 56–63, 74, 87, 181, 279

Nakamoto, Satoshi: Bitcoin and, xvi–xvii, xxii–xxvi, 18, 46, 119, 176; community and, 171; identity of, 176; mining and, 48, 204; original code of, 171; Satoshi Bones, 78; Satoshi denomination, 46, 216–17, 223; Satoshi Dice, 147–48, 224; white paper of, 176, 192
Namecoin, 224, 242, 247–48, 252, 257, 263, 265, 270–71, 274
Naor, Moni, xv–xvi, xix
Nash equilibrium, 43
National Institute of Standards and Technology (NIST), 26, 110, 196, 229–30
natural phenomena, 230–31
NBA draft lottery, 227
negative externalities, 198
NetCash, xviii–xix
Netscape, xii
network layer, 149–51
networks: algorithms and, 67, 69; attacks and, 69; Bitcoin mechanics and, 66–72; BOINC and, 198; bootstrapping and, 47–48; deanonymization and, 149–51; double spending and, 68–69; flooding algorithm and, 67–69; gossip protocol and, 67; hard forks and, 47, 73–75, 135, 172–73, 241, 252, 266, 270; lightweight, 71–72; orphan block and, 36, 46, 134; parameterizable cost and, 42–45; peer-to-peer, xi, xiv, 28–30, 32, 36, 42, 46–50, 59, 66–67, 96–97, 117, 128, 149, 151, 155, 176, 261; Simplified Payment Verification (SPV) and, 71, 190, 195, 218, 223, 247, 261–63, 277; size of, 69–70; social, 27–29; soft forks and, 47, 73–74, 159, 172–73, 241, 256, 260; storage requirements and, 70–71; Tor and, 143, 150, 153, 157, 167, 179–81; transaction fees and, 97–98; whitelists and, 59, 67
New York Department of Financial Services (NYDFS), 186–89
New York Knicks, 227–29
nodes: altcoins and, 242, 247, 260–62; anonymity and, 149–50; Bitcoin mechanics and, 59, 66–75; consensus and, 28–40, 46–50, 168; decentralization and, 28–49; exchanges and, 93; full, 217, 247, 277; honest, 29, 34–38, 43, 48–49; master, 66; Merkle trees and, 12–14, 64–65, 92–93, 105–7, 201–2, 204, 217, 255, 269; mining and, 42, 104, 111, 113, 117, 125, 130, 134, 190, 203, 210; parent, *13*; payments and, 97–98; platform issues and, 217, 219; random, 33–35, 38, 40–41; Sybil attacks and, 32–33, 40–41; transaction pools and, 30
nonces: altcoins and, 257; Bitcoin mechanics and, 65; commit function and, 6; cryptography and, 6–8; decentralization and, 41–44; mining and, 104–7, 111–13, 124, 199, 202; platform issues and, 232; random, 6–7, 41, 199, 202
nothing-at-stake problem, 209–10
NSA, 138

Obama, Barack, 236
offline guessing, 82
Ohta, Kazuo, xvi

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

Okamoto, Tatsuaki, xvi
one-way pegs, 245
online guessing, 82
OpenAssets, 221–23, 241
open protocols, 71, 174, 241
order books, 231, 236, 240–41, 268
orphan block, 36, 46, 134
overclocking, 113, 115
overlay currencies, 218–19, 247

parameterizable cost, 42–45
Pareto improvement, 183
partial hash-preimage puzzle, 191, 193
passphrases, 81–82
passwords, 82–83, 86, 88, 103, 152, 193, 195, 264
patents, xvi, 214
Patricia tree, 269
Paxos, 31, 50
PayCoins, 24, 52
paying for a proof, 276–77
payments, vii; altcoins and, 242, 244, 251, 261, 263, 267–69; anonymity and, 140–42, 146–47, 154–55, 158–59; block chains and, 97–98 (*see also* block chains); community and, 174, 178–80; cryptography and, 86; decentralization and, 34–37, 39, 48, 274, 276–77, 281–82; deposits and, 96–97 (*see also* deposits); disputes and, 60–61, 214, 238, 274, 278–80, 283–85; exchanges and, 89, 91; fiat currencies and, 94–97; HTML and, 94–96; joint, 53; lock time and, 63–64; mechanics of, 53; micropayments and, xiv, 62–64, 268; mining and, 122, 126–27, 131–32, 206–7; nodes and, 97–98; peer-to-peer networks and, 96–97; platform issues and, 237–38; prediction markets and, 237–38; scripts and, 62–64; services for, 94–99; settlements and, 96, 221, 237–38, 242; Simplified Payment Verification (SPV) and, 71, 190, 195, 218, 223, 247, 261–63, 277; smart contracts and, 64, 219, 263–70; stakeholders and, 174; storage and, 86, 94–99; timestamps and, 31, 59, 63, 213–17, 222, 277; transaction fees and, 25, 39–42, 45–46, 54 (*see also* transaction fees)
PayPal, ix, xii–xiii, 72, 285
pay-per-share model, 126–27
Pay-to-Script-Hash (P2SH) address, 59–60, 74, 218, 221
Peercoin, 208–10
peer-to-peer networks: altcoins and, 261; anonymity and, 149, 151, 155; Bitcoin mechanics and, 59, 66–67; decentralization and, 28–32, 36, 42, 46–50; mining and, 117, 128; parameterizable cost and, 42–45; payments and, 96–97; Satoshi white paper and, 176
PGP, xvii
phishing, 283
physical property, 223
platform issues: algorithms and, 218–19; append-only logs and, 22, 51, 213–19, 247; attacks and, 214, 216, 233–34; block chains and, 217–19, 223–24, 232; clairvoyance and, 214–16; colored coins and, 221–24, 277; consensus and, 218–19; cryptocurrencies and, 234, 238–39, 241; data feeds and, 234–39; data structures and, 213; deposits and, 226, 234; domain names and, 29, 223–24, 248, 257; double spending and, 218; escrow and, 227; forks and, 233, 241; fungibility and, 219; hash functions and, 213–14, 217, 232; identity and, 216; illicit content and, 217–18; lotteries and, 224–34, 241; metadata and, 220–22; miners and, 216–19, 222–23, 232–34, 238, 240; nonces and, 232; OpenAssets and, 221–23, 241; order books and, 231, 236, 240–41, 268; overlay currencies and, 218–19, 247; payments and, 237–38; privacy and, 219; private keys and, 216–17, 239; public keys and, 214–17, 236, 239; puzzles and, 232; SHA-256 and, 217; signatures and, 214, 216–17, 220, 226, 229, 238–39; smart property and, 219–24; third parties and, 223; timestamping and, 213–14; transaction fees and, 216–18, 233, 240; unspendable outputs and, 217
Poisson distribution, 124, *125*
Poisson process, 43–44
politics, vii, 220, 253, 286; blocks and, 66, 75; capital controls and, 178; crime and, 142–43, 178–81, 240; cryptocurrencies and, 198; cryptography and, 285; decentralization and, 282, 285; fiat currencies and, 183, 188; law enforcement and, 1, 135, 143, 149, 168, 178–81, 283; legal issues and, 204; military draft lottery and, 227–29; selling votes and, 204; Silk Road and, 165, 179–81, 189
Ponzi schemes, 89–90
Popper, Nathaniel, xxii
pornography, 217–18
prediction markets: altcoins and, 263, 268; arbitration and, 238–39; CreateMarket and, 236–38; data feeds and, 234–39; decentralization and, 236–37, 279–82; order books and, 231, 236, 240–41, 268; payments and, 237–38; platform issues and, 234–41; power of, 235; profits from, 234–38, 240; real-world data feeds and, 234–41; settlement and, 237–38

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

prefix tree, 269
pre-mining, 244–45
price ceilings, 245
Primecoin, 200–203
prime numbers, 84–85, 163, *199*, 200–201
privacy: anonymity and, 138–44, 149–54, 159, 164, 166–67; community and, 175, 189; cryptography and, 20; decentralization and, 284; exchanges and, 91–94; NSA and, 138; platform issues and, 219; pseudonymity and, 32, 46, 139–44, 152, 164–65, 176, 180, 280; storage and, 77–81; Tor and, 143, 150, 153, 157, 167, 179–81
private keys, 18; altcoins and, 246–47, 250; anonymity and, 144, 156; decentralization and, 273, 276, 283; exchanges and, 91; mining and, 205–6, 210; platform issues and, 216–17, 239; scripts and, 58; storage and, 76–78, 80–83, 86; timestamps and, 216–17
profits: altcoins and, 245, 248, 253, 256, 270; attacks and, 233; Bitcoin investment and, 100; day traders and, 231; mining and, 45, 47, 105–6, 110, 112–13, 116–18, 124, 131, 132–36, 190, 197, 205; Ponzi schemes and, 89–90; prediction markets and, 234–38, 240
progress free puzzles, 191, 199, 201
proof of burn, 59, 158, 217, 245–46
proof of clairvoyance, 214–16
proof of deposit, 209
proof of liabilities, 91–94
proof of membership, 13–14
proof of nonmembership, 14–15
proof of reserve, 91, 93–94
proof of retrievability, 201
proof of stake, 41, 206–11
proof of storage, 201–3
proof of work: altcoins and, 243, 257, 260–63, 270; anonymity and, 157; decentralization and, 38–45, 50; economic issues and, 203; incentives and, 38–45; mining and, 40–42, 193, 195, 198–203, 208, 211; negative externalities and, 198; previous distributed computing projects and, 198–99; Primecoin and, 200–203; public good and, 203; puzzle adaption and, 199–200; spare cycles and, 198
proportional model, 127
protocol limitations: algorithms and, 72–73; attacks and, 73; improvements and, 72–75; metadata and, 74
pseudonymity, 32, 46, 139–44, 152, 164–65, 176, 180, 280
public good, 203
public keys, xiii; altcoins and, 265; anonymity and, 139, 143–44, 163; Boolean validation and, 15; community and, 175; compression and, 18; consensus and, 29; decentralization and, 29, 34, 273, 276; as identities, 18–24; mining and, 107, 202, 204–6; platform issues and, 214–17, 236, 239; scripts and, 55–60; signatures and, 214; stealth addresses and, 144; storage and, 78–83; unforgeability and, 16; vanity address generation and, 78; verification and, 14–18
pump-and-dump scams, 244–45
puzzle friendliness, 8–10, 41, 198
puzzle-ID, 9
puzzles: algorithmically generated, 200; altcoins and, 248–56, 270; alternative, 190–211; anonymity and, 160; Bitcoin mechanics and, 64; block-discarding attacks and, 204–5; cryptography and, 2, 8–10; Cuckoo Cycle, 195; Cunningham chain and, 200–201; decentralization and, 41–43, 46–47, 50; equiprobable solution space and, 199; inexhaustible space and, 199–200; memory-bound, 193, 195, 211; memory-hard, 193–96, 211, 248, 270; mining and, 64, 107, 119, 122, 190–211, 248–56, 270; nothing-at-stake problem and, 209–10; platform issues and, 232; proof of retrievability and, 201; sabotage attacks and, 205–6; scrypt and, 193–96, 202, 211, 248, 256; stake-grinding attacks and, 209–10; trends in, 256; vigilante attacks and, 205

QR codes, 77–78

random oracle model, 10
Reddit, 139
refunds, 63, 185, 258–59
regulation: anti-money laundering (AML) and, 181–83; antitrust and, 186; asymmetric information and, 184; bad reputation of, 183; banks and, 90–91, 99, 168, 175, 178; collusion and, 186; crime and, 142–43, 178–81, 240; fiat currencies and, 183; government-issued ID and, 99; justification of, 183–86; law enforcement and, 1, 135, 143, 149, 168, 178–81, 283; legal issues and, 179, 181, 183, 186; lemons market and, 184–86; libertarians and, 175, 188; licenses and, 170, 186–89; mandatory reporting and, 182–83; market fixes and, 184–86; money laundering and, xxvi, 142, 166, 181–83; Pareto improvement and, 183; Silk Road and, 165, 179–81, 189
replace-by-fee, 69
Request for Comments (RFC), 174
Ripple, 242
Rivest, Ron, xx
RSA, xii, xx, 163

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

sabotage attacks, 205–6
Satoshi Bones, 78
Satoshi denomination, 46, 216–17, 223
Satoshi Dice, 147–48, 224
scripts: applications of, 60–64; beacons and, 233–34; Bitcoin mechanics and, 55–64; double spending and, 62–63; escrow transactions and, 60–61; executing, 57–58; green addresses and, 61–63; lock time and, 63–64; micropayments and, 63–64; P2SH, 59–60, 74, 218, 221; payments and, 62–64; in practice, 58–59; private keys and, 58; proof of burn and, 59, 158, 217, 245–46; public keys and, 55–60; smart contracts and, 64, 219, 263–70; third parties and, 60–61; transaction fees and, 62; verification and, 86; whitelist, 59, 67
scriptSig, *54*, 55–57, 226, 254–59
Scroogecoin, 22–25, 27, 29–30, 39, 52–53, 65
scrypt, 193–96, 202, 211, 248, 256
search puzzles, 8–9
secret sharing, 83–87
secure multiparty computation: fairness and, 225–27; lotteries and, 224–34, 241; online coin flipping and, 225; platform issues and, 224–34, 241
security: 51 percent attacker and, 48–49, 128–30, 131–32, 197, 208–11 (*see also* attacks); append-only ledger and, 22–23, 51, 213–19, 247; base-58 notation and, 77, 83; beacons and, 229–34; challenges of real-world, 283–84; collision resistance and, 2–5; compression function and, 9–10, 18, 111–12; counterfeiting and, 1, 220; credit cards online and, xi–xiii; cryptography and, 1 (*see also* cryptography); decentralization and, 279–80, 283–84; disputes and, 60–61, 214, 238, 274, 278–80, 283–85; double spending and, xiv–xvi, 22 (*see also* double spending); encryption and, xi, 19, 84, 88, 179, 192; equivocation and, 1; Ethereum and, 266–67; exchanges and, 274–75; forgery and, 15–18, 25, 34, 67, 240–41; genesis block and, 12, 77, 171–72, 201, 210, 219, 242; Goofycoin and, 21–24; hackers and, 86, 90, 152, 165, 203, 218, 267, 275; key stretching and, 82; ledgers and, xx–xxiii, 22, 24, 27–28, 30, 32, 47, 49, 51–53, 141, 164, 268–69; lotteries and, 33, 63, 224–34, 241; merge mining and, 256–57; money laundering and, xxvi, 142, 166, 181–83; NSA and, 138; passphrases and, 81–82; passwords and, 82–83, 86, 88, 103, 152, 193, 195; Ponzi schemes and, 89–90; private keys and, 18 (*see also* private keys); proof of membership and, 13–14; proof of nonmembership and, 14–15; public keys and, 15–24, 29 (*see also* public keys); QR codes and, 77–78; randomness and, 20; random oracle model and, 10; Scroogecoin and, 22–25, 27, 29–30, 39, 52–53, 65; secret keys and, 76, 79–80, 83–87, 198; SET architecture and, xii–xiii; smart contracts and, 266–67; storage and, 76, 79–80, 83–85, 198; tampering and, 1, 5, 11–13, 83, 213, 230, 247; theft and, 20, 34, 48, 76–77, 81, 84–87, 144, 155, 157, 181, 206, 238, 260, 262, 279, 283; timestamps and, 216–17; unforgeability and, 15–17; usability and, xiii; wallets and, 28, 62, 71, 77–88, 94–96, 98, 139, 141, 144–48, 151–55, 165, 187, 247, 251–52; zero-knowledge proofs and, 1, 160–64, 166, 229
selfish mining, 134
SET architecture, xii–xiii
SETI@home, 198–200
settlements, 96, 221, 237–38, 242
SHA-256: altcoins and, 250, 253, 256; Bitcoin mechanics and, 57, 73; compression function and, 9–10, 111–12; cryptography and, 9–10; hash function of, 9–10, 57, 73, 82, 110–16, 119, 120, 122, 191–202, 217, 250, 253, 256; initialization vector (IV) and, 9, *10*; Merkle-Damgård transform and, 9–10, 12; mining and, 110–16, 119–22, 191–202, 208, 250, 253, 256; platform issues and, 217; storage and, 82
SHA-512, 110
Shamir, Adi, xx
shared spending, 145, 147
sidechains, 260–63, 270, 278
side channels, 140, 153, 157–58, 164–65
signatures: altcoins and, 246, 258–59; anonymity and, 142, 156, 162; bitcoin mechanics and, 52–61, 70–73; blind, xv, 142; blocks and, 205–6; cryptography and, 1, 15–26, 34, 80, 220, 229, 273; decentralization and, 34, 48, 273–74, 276, 279; digital, 1, 15–21, 26, 34, 80, 220, 229, 273; Elliptic Curve Digital Signature Algorithm (ECDSA) and, 17–19, 26, 73, 80, 144, 216, 273, 276; generateKeys and, 15–16, 19, 80–81; Guy Fawkes scheme and, 214; handwritten, 15; mining and, 104, 205–6, 210; multiple, 56–63, 74, 87, 181, 279; platform issues and, 214, 216–17, 220, 226, 229, 238–39; public keys and, 80, 214 (*see also* public keys); sabotage attacks and, 205–6; storage and, 80, 86–87; threshold, 86–87; unforgeability and, 15–17; verification and, 56, 58
Silk Road, 165, 179–81, 189
Simple Mail Transfer Protocol (SMTP), 27–28
Simplified Payment Verification (SPV), 71, 190, 195, 218, 223, 247, 261–63, 277
smart contracts, 64, 219; altcoins and, 263–70; block chains and, 263–70; enforcement and, 264–

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

65; Ethereum and, 263–70; virtual machines and, 265–66, 270
smart property, 219–24, 257, 268, 273–74, 281–85
soccer, 215
social networks, 27–29
soft forks, 47, 73–74, 159, 172–73, 241, 256, 260
Solidity, 265
sorted Merkle tree, 14
spam, xix
spare cycles, 198
spoofing, 273
SPV proofs, 261–63
stake-grinding attacks, 209–10
stakeholders, 138, 173–75, 186, 203, 208, 244
standards document, 9
stealth addresses, 144
Stellar, 242
storage: algorithms and, 81; attacks and, 82; base-58 notation and, 77, 83; block chains and, 76, 79, 81–82, 86; cold, 79–83, 87; exchanges and, 87–94; hash functions and, 78–79, 82; hot, 79–83, 90; identity and, 76; message digests and, 4–5, 17; networks and, 70–71; passphrases and, 81–82; payments and, 86, 94–99; Ponzi schemes and, 89–90; private keys and, 76–78, 80–83, 86; proof of retrievability and, 201; public keys and, 78–83; QR codes and, 77–78; secret keys and, 76, 79–80, 83–87, 198; SHA-256 and, 82; signatures and, 80, 86–87; simple local, 76–79; splitting/sharing keys and, 83–87; StorJ and, 282; vanity addresses and, 78–79; verification and, 86; wallets and, 28, 62, 71, 77–88, 94–96, 98, 139, 141, 144–48, 151–55, 165, 187, 247, 251–52
StorJ, 282
Stornetta, W. S., xx
stylometry, 176
supply and demand, 99–101, 266
Suspicious Activity Report, 182
switching costs, 252
Sybil attack, 32–33, 40–41
Szabo, Nick, xxii, xxiv

tagging, 148–49
Tahoe-LAFS, xi
taint analysis, 141
tamper-evident logs, 11–12, 83
tampering, 1, 5, 11–13, 83, 213, 230, 247
tamper-resistant devices, 83
targets: altcoins and, 254–55, 262–63, 270; anonymity and, 160; cryptography and, 8; decentralization and, 41–45; hash functions and, 8, 41–45, 105–6, *113*, 125, 160, 191, 196, 202–6, 254–55, 262–63, 270; mining and, 105–6, *113*, 125, 191, 196, 202–6, 254
Tesla, Nikola, 252
third parties: altcoins and, 250–51; decentralization and, 274; escrow transactions and, 60; platform issues and, 223; scripts and, 60–61
Three Gorges Dam, 122
threshold signatures, 86–87
time-memory trade-offs, 194–95
timestamps, xxiv, 31, 59, 63, 213–17, 222, 277
Tinkerbell effect, 169, 244
Tor, 143, 150, 153, 157, 167, 179–81
transaction fees: altcoins and, 266; anonymity and, 140, 154, 156, 164; blocks and, 65, *66*, 105; decentralization and, 39–40, 45–46, 277; definition of, 97; greed and, 25; as incentive mechanism, 39–40; mining and, 54, 97–98, 131, 136, 203, 211; networks and, 97–98; platform issues and, 216–18, 233, 240; replace-by-fee and, 69; scripts and, 62; setting, 98; timestamping and, 216
transaction graph analysis, 149, 151, 164–66, 219, 222, 269
transactions: 51 percent attacker and, 48–49, 128–30, 131–32, 197, 208–11; append-only ledger and, 22–23, 51, 213–19, 247; Bitcoin mechanics and, 51–55; block chains and, 97–98 (*see also* block chains); change address and, 52–53, 62, 145–47, 268; coinbase, 65–66, 74, 88, 94, 105–7, 125, 204–6, 219, 254–56; CoinJoin and, 156–58; contesting a transfer and, 261–62; disputes and, 60–61, 214, 238, 274, 278–80, 283–85; efficiency and, 162–63; escrow, 60–64, 180–81, 227, 247, 260, 263, 268, 279; green addresses and, 61–63; HTML and, 94–96; inputs and, 54; legal issues and, 179 (*see also* legal issues); listening for, 104; mandatory reporting and, 182–83; metadata and, 53–54, 64; micropayments and, xiv, 63–64, 268; outputs and, 54; P2SH, 60, 74; price ceilings and, 245; proof of burn and, 59, 158, 217, 245–46; replace-by-fee, 69; scripts and, 55–64; settlements and, 96, 221, 237, 242; signatures and, 1 (*see also* signatures); Simplified Payment Verification (SPV) and, 71, 190, 195, 218, 223, 247, 261–63, 277; smart contracts and, 64, 219, 263–70; syntax and, 53; tagging and, 148–49; third parties and, 60–61, 223, 250–51, 274; zero-confirmation, 36, 69. *See also* payments
Tromp, John, 195
trust, 280
tumblers, 152
Turing completeness, 263
Twitter, 215

For general queries contact webmaster@press.princeton.edu.

© Copyright Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

Ulbricht, Ross, 180–81
unforgeability, 15–17
unlinkability, 81, 139–40, 144, 151, 157–59, 164

valid blocks: altcoins and, 253; Bitcoin mechanics and, 68, 73–74; community and, 168; decentralization and, 30, 39, 48; mining and, 73–74, 105–6, 111–13, 125–27, 133–34, 199, 204–5, 208, 210
vanity addresses, 78–79
verification: altcoins and, 260–62, 268; Bitcoin mechanics and, 53, 56, 58, 71; cryptography and, 14–18; efficient, 53; mining and, 191, 195–96, 203; public keys and, 14–18; scripts and, 86; signatures and, 56, 58; Simplified Payment Verification (SPV) and, 71, 190, 195, 218, 223, 247, 261–63, 277; storage and, 86
Verisign, xii–xiii
Vietnam War, 227–29
vigilante attacks, 205
Virtual Currency Business Activity, 187
virtual machines, 265–66, 270
Visa, 72, 285
VisaCash, xviii

wagers, 148, 239, 267, 281
wallets, xviii, 187; altcoins and, 247, 251–52; anonymity and, 139, 141, 144–48, 151–55, 165; bank regulation and, 90–91; bank runs and, 89–90; base-58 notation and, 77, 83; brain, 81–83, 87; decentralization and, 28; exchanges and, 87–94; hierarchical, 80–81; hot, 79, 84; Instawallet and, 62; mixing and, 151–52; paper, 83; passphrases and, 81–82; payment services and, 94–96; Ponzi schemes and, 89–90; QR codes and, 77–78; SPV nodes and, 71; stealth addresses and, 144; transaction fees and, 98; two-factor security and, 86
wattage, 121–22, 198, 203
whitelist scripts, 59, 67
whitepapers, xxiv, 166, 271
Wikileaks, 138, 143–44
Wikipedia, xxiv
Wired UK magazine, 138
withdrawals, 88–90, 151–52, 265–66
World Cup, 215

X11, 196–97

Y2K bug, xiii

Zerocash, 143, 159, 163–66, 282
Zerocoin: altcoins and, 260; anonymity and, 143, 159–66; decentralization and, 281–82
zero-confirmation transactions, 36, 69
zero-knowledge proofs, xvi, 1, 160–64, 166, 229
Zetacoin, 250

For general queries contact webmaster@press.princeton.edu.