Bitcoin and Cryptocurrency Technologies

Arvind Narayanan, Joseph Bonneau, Edward Felten,
Andrew Miller, Steven Goldfeder

Draft — Oct 6, 2015

Feedback welcome! Email bitcoinbook@lists.cs.princeton.edu

Introduction to the book

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

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

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

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

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

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

After reading this book, you'll know everything you need to be able to separate fact from fiction when reading claims about Bitcoin and other cryptocurrencies. You'll have the conceptual foundations you need to engineer secure software that interacts with the Bitcoin network. And you'll be able to integrate ideas from Bitcoin into your own projects.

Chapter 1: Introduction to Cryptography & Cryptocurrencies

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

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

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

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

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

1.1 Cryptographic Hash Functions

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

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

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

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

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

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

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

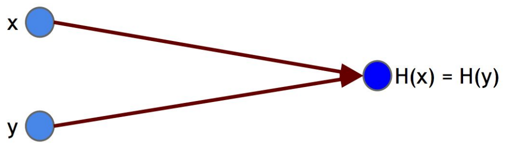


Figure 1.1 A hash collision. *x* and *y* are distinct values, yet when input into hash function *H*, they produce the same output.

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

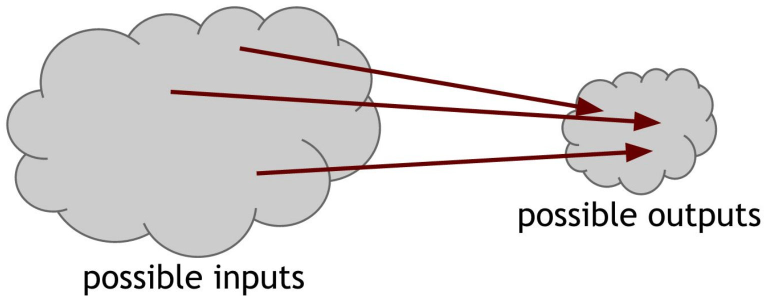


Figure 1.2 Because the number of inputs exceeds the number of outputs, we are guaranteed that there must be at least one output to which the hash function maps more than one input.

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

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

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

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

Consider, for example, the following hash function:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Commitment scheme. A commitment scheme consists of two algorithms:

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

We require that the following two security properties hold:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Search puzzle. A search puzzle consists of

- a hash function, *H*,
- a value, *id* (which we call the ***puzzle-ID***), chosen from a high min-entropy distribution
- and a target set *Y*

A solution to this puzzle is a value, *x*, such that

$$H(id \parallel x) \in \boldsymbol{Y}.$$

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

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

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

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

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

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

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

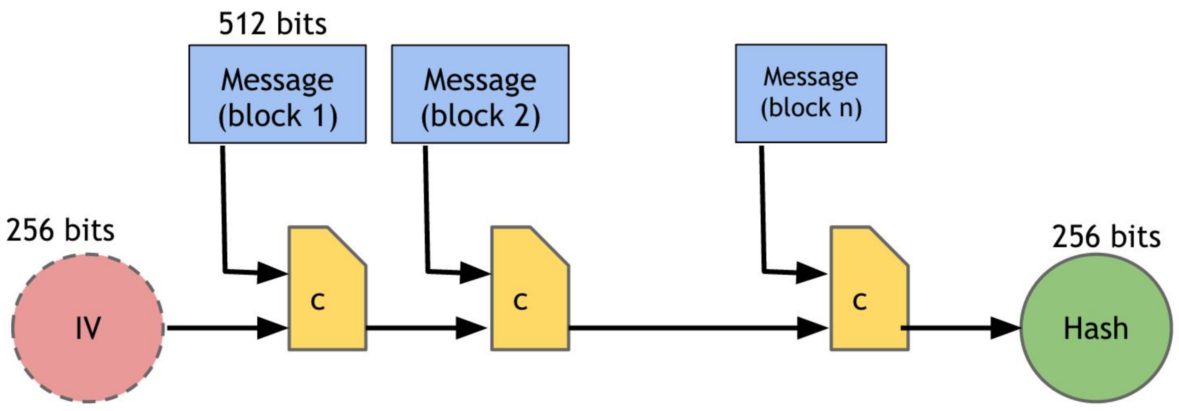


Figure 1.3: SHA-256 Hash Function (simplified). SHA-256 uses the Merkle-Damgard transform to turn a fixed-length collision-resistant compression function into a hash function that accepts arbitrary length inputs.

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

1.2 Hash Pointers and Data Structures

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

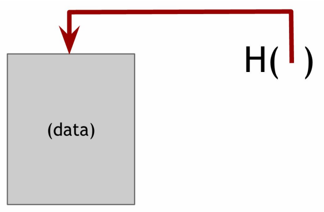


Figure 1.4 Hash pointer. A hash pointer is a pointer to where data is stored together with a cryptographic hash of the value of that data at some fixed point in time.

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

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

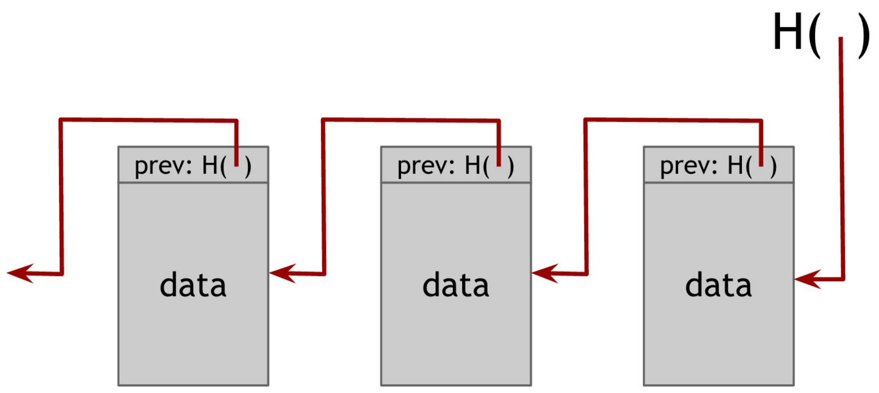


Figure 1.5 Block chain. A block chain is a linked list that is built with hash pointers instead of pointers.

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

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

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

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

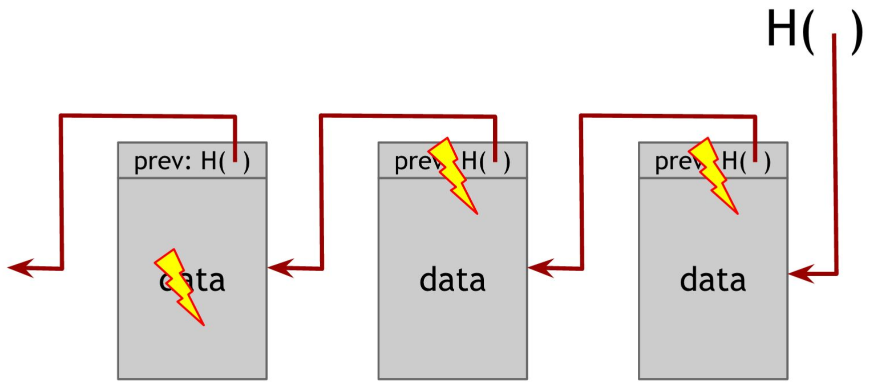


Figure 1.6 Tamper-evident log. If an adversary modifies data anywhere in the block chain, it will result in the hash pointer in the following block being incorrect. If we store the head of the list, then even if the adversary modifies all of the pointers to be consistent with the modified data, the head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

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

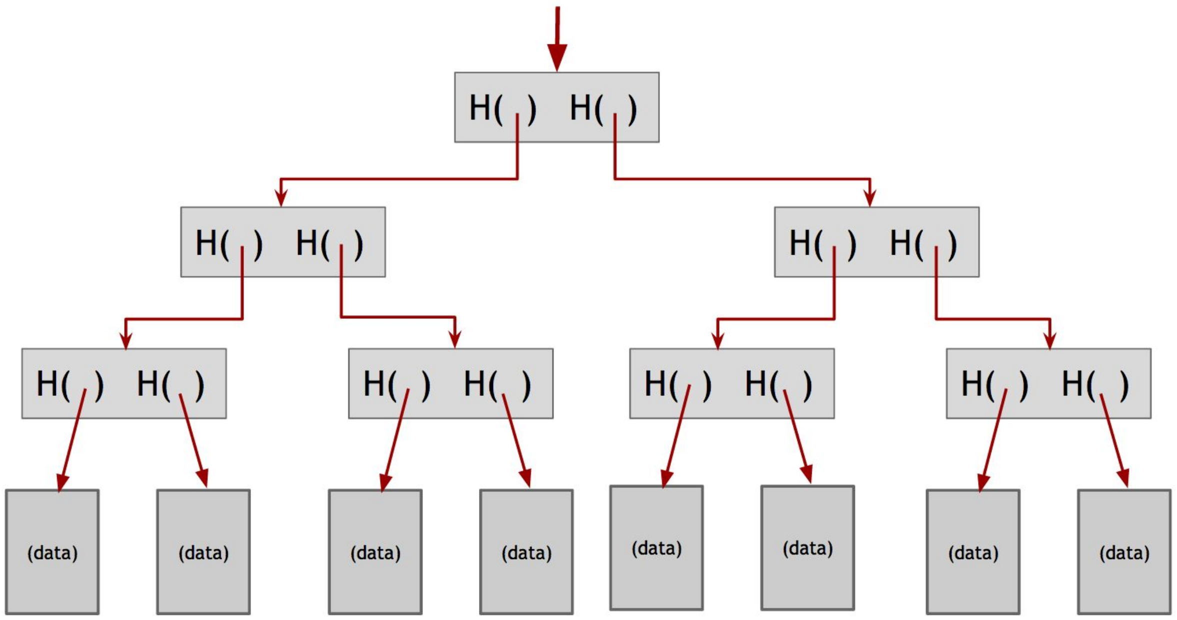


Figure 1.7 Merkle tree. In a Merkle tree, data blocks are grouped in pairs and the hash of each of these blocks is stored in a parent node. The parent nodes are in turn grouped in pairs and their hashes stored one level up the tree. This continues all the way up the tree until we reach the root node.

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

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

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

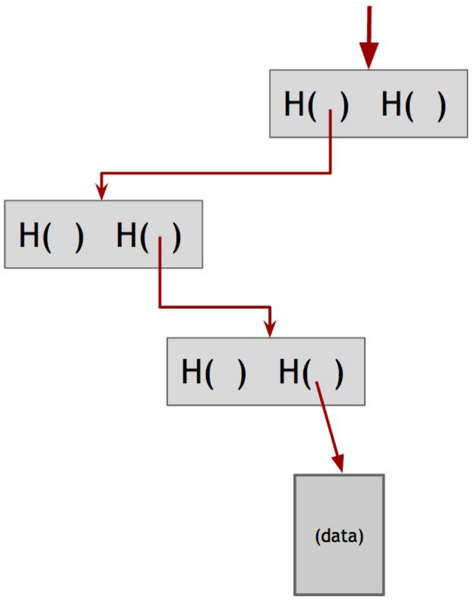


Figure 1.8 Proof of membership. To prove that a data block is included in the tree, one only needs to show the blocks in the path from that data block to the root.

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

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

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

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

1.3 Digital Signatures

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

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

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

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

We require that the following two properties hold:

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

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

Let us now examine the two properties that we require of a digital signature scheme in more detail. The first property is straightforward — that valid signatures must verify. If I sign a message with *sk*, my secret key, and someone later tries to validate that signature over that same message using my public 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 gets to see 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, there is an adversary who claims that he can forge signatures and a challenger that will test this claim. The first thing we do is we 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 to 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-life attacker would likely be able to see valid signatures from their would-be victim on a number of different documents. And maybe the attacker could even manipulate the victim into signing innocuous-looking documents if that's useful to the attacker.

To model this in our game, we're going to allow the attacker 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 attacker to try 1 million guesses, but not 2^80 guesses[2].

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

[2] In asymptotic terms, we allow the attacker to try a number of guesses that is a polynomial function of the key size, but no more (e.g. the attacker cannot try exponentially many guesses).

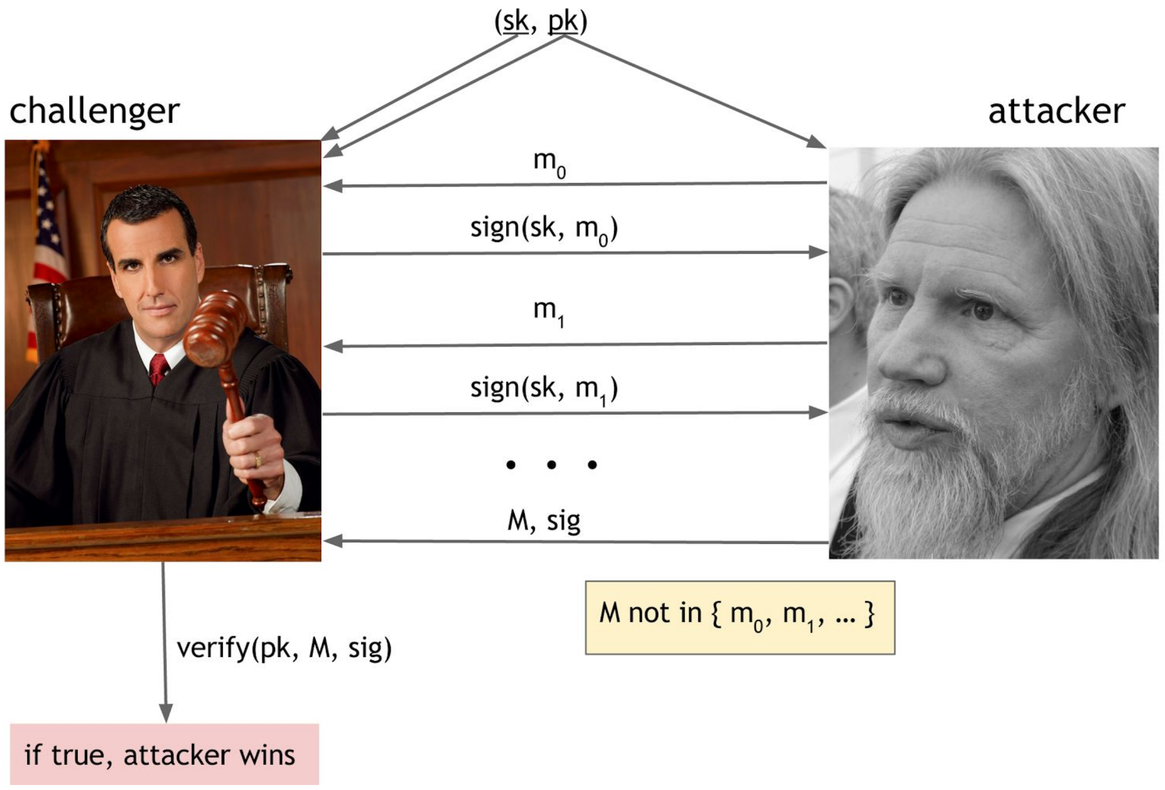


Figure 1.9 Unforgeability game. The adversary and the challenger play the unforgeability game. If the attacker is able to successfully output a signature on a message that he has not previously seen, he wins. If he is unable, the challenger wins and the digital signature scheme is unforgeable.

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. There are a number of practical things that we need to do to turn the algorithmic idea into a digital signature mechanism that can be implemented in practice. 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 really can't be underestimated 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 discussed before, 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 that was at the end of a block chain, the result is that you would effectively be digitally signing the that entire block chain.

ECDSA. Now let's get into the nuts and bolts. Bitcoin uses a particular digital signature scheme that's called the Elliptic Curve Digital Signature Algorithm (ECDSA). ECDSA is a U.S. government standard, an update of the earlier DSA algorithm adapted to use elliptic curves. These algorithms have received considerable cryptographic analysis over the years and are generally believed to be secure.

More specifically, Bitcoin uses ECDSA over the standard elliptic curve "secp256k1" which is estimated to provide 128 bits of security (that is, it is as difficult to break this algorithm as performing 2^{128} symmetric-key cryptographic operations such as invoking a hash function). While this curve is a published standard, it is rarely used outside of Bitcoin, with other applications using ECDSA (such as key exchange in TLS for secure web browsing) typically using the more common "secp256r1" curve. This is just a quirk of Bitcoin, as this was chosen by Satoshi 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 there's some complicated math involved, and understanding it is not necessary for any other content in 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 while ECDSA can technically only sign messages 256 bits long, this is not a problem in practice: 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 of randomness will likely leak your key. It makes intuitive sense that if you use bad randomness in generating a key, then the key you generate will likely not be secure. But it's a quirk of ECDSA[3] that, even if you use bad

[3] For those familiar with DSA, this is a general quirk in DSA and not specific to the elliptic curve variant.

randomness just in making a signature, using your perfectly good key, that also will leak your private key. 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, and 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'll 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 equate that 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* is saying the message. You can literally think of a public key as kind of 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. In order for someone to speak for the identity *pk*, they 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. *pk* is the new public identity that you can use, and *sk* is the corresponding secret key that only you know and lets you speak for 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 in order to verify that a message comes from your identity, one will have to check (1) that *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*.[4] You can generate a fresh identity that looks random, that looks like a face in the crowd, and that only you can control.

Decentralized identity management. This brings us to the idea of decentralized identity management. Rather than having a central authority that you have to go to in order to register as a user in a system, you can register as a user all 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 make 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 make a new identity, use it just for a little while, and then throw it away. All of these things are possible with decentralized identity management, and this is the way Bitcoin, in fact, does 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 that's really just a hash of a public

[4] Of course, once you start making statements using this identity, these statements may leak information that allows one to connect *pk* to your real world identity. We will discuss this in more detail shortly.

key. It’s an identity that someone made up out of thin air, as part of this decentralized identity management scheme.

Sidebar. 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. We are for all intents and purposes 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 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 it is crucial to use a good source of randomness when generating keys to ensure that practical guarantees match the theoretical ones.

On first glance, it may seems 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 infer things about your real-world identity. An observer can link together these things over time, and make inferences that lead them to conclusions such 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 the entirety of Chapter 6 to it.

1.5 A Simple Cryptocurrency

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’ll discuss two very simple cryptocurrencies. Of course, it's going to require much of the rest of the book to spell out all the implications of how Bitcoin itself works.

GoofyCoin

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

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

The second rule of GoofyCoin is that whoever owns a coin can transfer it on 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 signed by him exists, 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.

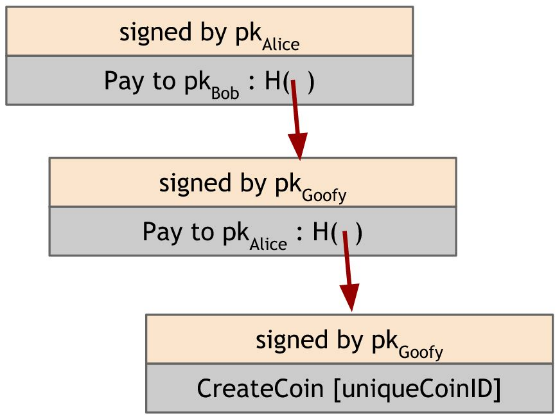


Figure 1.10 GoofyCoin coin. Shown here is a coin that's been created (bottom) and spent twice (middle and top).

Once Alice owns the coin, she can spend it in turn. To do this she creates a statement that says, “Pay this coin 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 would 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]”.

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 saying, “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 of the 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 very same coin to Chuck. To Chuck, it would appear that it is 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 very similar to Bitcoin, but because it is insecure it won’t cut it as a cryptocurrency.

ScroogeCoin

To solve the double-spending problem, we’ll design another cryptocurrency, which we’ll call 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 the transactions that have happened. The append-only property ensures that any data written to this ledger will remain forever. If the ledger is truly append-only, we can use it to defend against double-spending by requiring all transactions to be written the ledger before they are accepted. That way, it will be publicly visible if coins were previously sent to a different owner.

To implement this append-only functionality, Scrooge can build a block chain (the data structure we discussed before) which he will digitally sign. It’s a series of data blocks, each with one transaction in it (in practice, as an optimization, we’d really put multiple transactions into 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 of the data in this entire structure, and publishes the signature along with the block chain.

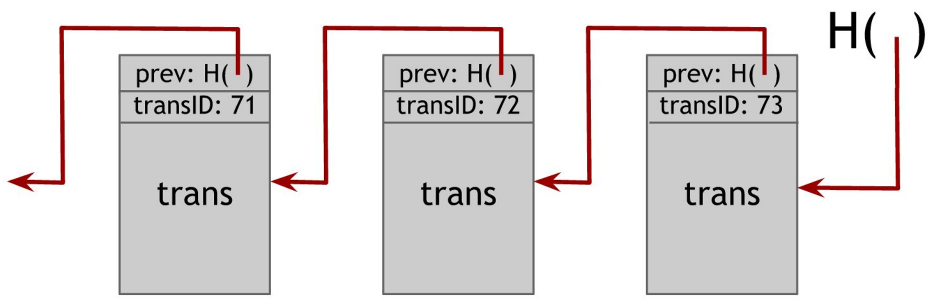
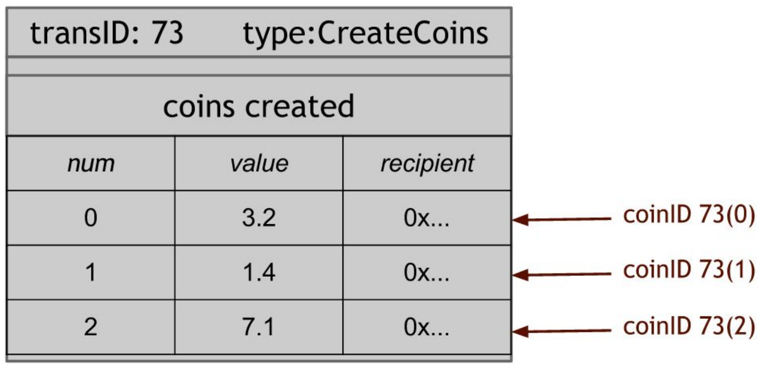


Figure 1.11 ScroogeCoin block chain.

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 it appears in. 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 to the history, or change an existing transaction, it will affect all of the 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 very easy for any two individuals to verify that they have observed the exact 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 that makes a new coin. With ScroogeCoin, we'll extend the semantics a bit to allow multiple coins to be created in one transaction.



transID: 73 type:CreateCoins

coins created

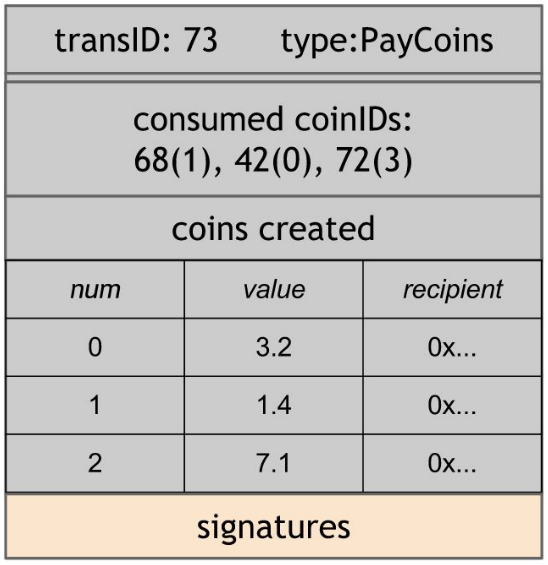
num	*value*	*recipient*
0	3.2	0x...
1	1.4	0x...
2	7.1	0x...

coinID 73(0)
coinID 73(1)
coinID 73(2)

Figure 1.12 CreateCoins transaction. This CreateCoins transaction creates multiple coins. Each coin has a serial number within 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 a bunch of 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 within that transaction.

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

The second kind of transaction is PayCoins. It consumes some coins, that is, 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 really okay with spending this coin.



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.**

The rules of ScroogeCoin say that PayCoins transaction is valid if four things are true:

- The consumed coins are valid, that is, they really were created in previous transactions.
- The consumed coins were not already consumed in some previous transaction. That is, that this is not a double-spend.
- 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 of the consumed coins.

If all of those conditions are met, then this PayCoins transaction is valid and Scrooge will accept it. He'll write it into the history 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 douple-spending transaction even if it is otherwise valid by the first three conditions.

Coins in this system are immutable — they are never changed, subdivided, or combined. Each coin is created, once, in one transaction and later consumed in some other 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 create 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 didn'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 of the 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 cartoon 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 in order to improve on ScroogeCoin and create a workable system is: can we descroogify 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 upon a single published block chain as the history of which transactions have happened. They must all agree on which transactions are valid, and which transactions have actually occurred. They also need to be able to assign IDs to things in a decentralized way. Finally, the minting of new coins needs to be controlled in a decentralized way. If we can solve all of those problems, then we can build a currency that would be like ScroogeCoin but without a centralized party. In fact, this would be a system very much like Bitcoin.

Further Reading

Steven Levy's *Crypto* is an enjoyable, non-technical 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.* 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, we refer you to:

Katz, Jonathan, and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* CRC Press, 2014.

For an introduction to applied cryptography, see:

Ferguson, Niels, Bruce Schneier, and Tadayoshi Kohno. *Cryptography engineering: design principles and practical applications*. John Wiley & Sons, 2012.

Perusing the NIST standard that defines SHA-2 is a good way to get an intuition for what cryptographic standards look like:

***FIPS PUB 180-4, Secure Hash Standards, Federal Information Processing Standards Publication.* Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 2008.**
http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

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.

Exercises

1. **Authenticated Data Structures.** You are designing SecureBox, an authenticated online file storage system. For simplicity, there is only a single folder. Users must be able to add, edit, delete, and retrieve files, and to list the folder contents. When a user retrieves a file, SecureBox must provide a proof that the file hasn't been tampered with since its last update. If a file with the given name doesn't exist, the server must report that — again with a proof.

We want to minimize the size of these proofs, the time complexity of verifying them, and the size of the digest that the user must store between operations. (Naturally, to be able to verify proofs, users must at all times store some nonzero amount of state derived from the folder contents. Other than this digest the user has no memory of the contents of the files she added.)

Here's a naive approach. The user's digest is a hash of the entire folder contents, and proofs are copies of the entire folder contents. This results in a small digest but large proofs and long verification times. Besides, before executing add/delete/edit operations, the user must retrieve the entire folder so that she can recompute the digest.

Alternatively, the digest could consist of a separate hash for each file, and each file would be its own proof. The downside of this approach is that it requires digest space that is linear in the number of files in the system.

Can you devise a protocol where proof size, verification time, and digest size are all sublinear? You might need a sub-protocol that involves some amount of two-way communication for the user to be able to update her digest when she executes and add, delete, or edit.

Hint: use the Merkle tree idea from Section 1.2.

2. **Birthday Attack.** Let H be an ideal hash function that produces an n-bit output. By ideal, we mean that as far as we can tell, each hash value is independent and uniformly distributed in $\{0,1\}^n$. Trivially, we can go through $2^n + 1$ different values and we are guaranteed to find a collision. If we're constrained for space, we can just store 1 input-output pair and keep trying new inputs until we hit the same output again. This has time complexity $O(2^n)$, but has $O(1)$ space complexity. Alternatively, we could compute the hashes of about $O(2^{n/2})$ different inputs and store all the input-output pairs. As we saw in the text, there's a good chance that some two of those outputs would collide (the "birthday paradox"). This shows that we can achieve a time-space trade-off: $O(2^{n/2})$ time and $O(2^{n/2})$ space.

1. (Easy) Show that the time-space trade-off is parameterizable: we can achieve any space complexity between $O(1)$ and $O(2^{n/2})$ with a corresponding decrease in time complexity.
2. (Very hard) Is there an attack for which the product of time and space complexity is $o(2^n)$? [Recall the little oh notation.]

3. **Hash function properties** (again). Let H be a hash function that is both hiding and puzzle-friendly. Consider $G(z) = H(z) \| z_{last}$ where z_{last} represents the last bit of z. Show that G is puzzle-friendly but not hiding.

4. **Randomness**. In ScroogeCoin, suppose Mallory tries generating (sk, pk) pairs until her secret key matches someone else's. What will she be able to do? How long will it take before she succeeds, on average? What if Alice's random number generator has a bug and her key generation procedure produces only 1,000 distinct pairs?

Bitcoin and Cryptocurrency Technologies

Arvind Narayanan, Joseph Bonneau, Edward Felten,
Andrew Miller, Steven Goldfeder

Draft — Oct 6, 2015

Chapter 2: How Bitcoin Achieves Decentralization

In this chapter, we will discuss decentralization in Bitcoin. In the first chapter we looked at the crypto basics that underlie Bitcoin and we ended with a simple currency that we called ScroogeCoin. ScroogeCoin achieves a lot of what we want in a ledger-based cryptocurrency, but it has one glaring problem — it relies upon the centralized authority called Scrooge. We ended with the question of how to decentralize, or de-Scrooge-ify, this currency, and answering that question will be the focus of this chapter.

As you read through this chapter, take note that the mechanism through which Bitcoin achieves decentralization is not purely technical, but it's a combination of technical methods and clever incentive engineering. At the end of this chapter you should have a really good appreciation for how this decentralization happens, and more generally how Bitcoin works and why it is secure.

2.1 Centralization vs. Decentralization

Decentralization is an important concept that is not unique to Bitcoin. The notion of competing paradigms of centralization versus decentralization arises in a variety of different digital technologies. In order to best understand how it plays out in Bitcoin, it is useful to understand the central conflict — the tension between these two paradigms — in a variety of other contexts.

On the one hand we have the Internet, a famously decentralized system that has historically competed with and prevailed against "walled-garden" alternatives like AOL's and CompuServe's information services. Then there's email, which at its core is a decentralized system based on the Simple Mail Transfer Protocol (SMTP), an open standard. While it does have competition from proprietary messaging systems like Facebook or LinkedIn mail, email has managed to remain the default for person-to-person communication online. In the case of instant messaging and text messaging, we have a hybrid model that can't be categorically described as centralized or decentralized. Finally there's social networking: despite numerous concerted efforts by hobbyists, developers and entrepreneurs to create alternatives to the dominant centralized model, centralized systems like Facebook and LinkedIn still dominate this space. In fact, this conflict long predates the digital era and we see a similar struggle between the two models in the history of telephony, radio, television, and film.

Decentralization is not all or nothing; almost no system is purely decentralized or purely centralized. For example, email is fundamentally a decentralized system based on a standardized protocol, SMTP, and anyone who wishes can operate an email server of their own. Yet, what has happened in the market is that a small number of centralized webmail providers have become dominant. Similarly, while the Bitcoin protocol is decentralized, services like Bitcoin exchanges, where you can convert

Bitcoin into other currencies, and wallet software, or software that allows people to manage their bitcoins may be centralized or decentralized to varying degrees.

With this in mind, let's break down the question of how the Bitcoin protocol achieves decentralization into five more specific questions:

1. Who maintains the ledger of transactions?
2. Who has authority over which transactions are valid?
3. Who creates new bitcoins?
4. Who determines how the rules of the system change?
5. How do bitcoins acquire exchange value?

The first three questions reflect the technical details of the Bitcoin protocol, and it is these questions that will be the focus of this chapter.

Different aspects of Bitcoin fall on different points on the centralization/decentralization spectrum. The peer-to-peer network is close to purely decentralized since anybody can run a Bitcoin node and there's a fairly low barrier to entry. You can go online and easily download a Bitcoin client and run a node on your laptop or your PC. Currently there are several thousand such nodes. Bitcoin *mining*, which we'll study later in this chapter, is technically also open to anyone, but it requires a very high capital cost. Because of this there has been a high degree of centralization, or a concentration of power, in the Bitcoin mining ecosystem. Many in the Bitcoin community see this as quite undesirable. A third aspect is updates to the software that Bitcoin nodes run, and this has a bearing on how and when the rules of the system change. One can imagine that there are numerous interoperable implementations of the protocol, as with email. But in practice, most nodes run the reference implementation, and its developers are trusted by the community and have a lot of power.

2.2 Distributed consensus

We've discussed, in a generic manner, centralization and decentralization. Let's now examine decentralization in Bitcoin at a more technical level. A key term that will come up throughout this discussion is ***consensus***, and specifically, ***distributed consensus***. The key technical problem to solve in building a distributed e-cash system is achieving distributed consensus. Intuitively, you can think of our goal as decentralizing ScroogeCoin, the hypothetical currency that we saw in the first chapter.

Distributed consensus has various applications, and it has been studied for decades in computer science. The traditional motivating application is reliability in distributed systems. Imagine you're in charge of the backend for a large social networking company like Facebook. Systems of this sort typically have thousands or even millions of servers, which together form a massive distributed database that records all of the actions that happen in the system. Each piece of information must be recorded on several different nodes in this backend, and the nodes must be in sync about the overall state of the system.

The implications of having a distributed consensus protocol reach far beyond this traditional application. If we had such a protocol, we could use it to build a massive, distributed key-value store, that maps arbitrary keys, or names, to arbitrary values. A distributed key-value store, in turn, would enable many applications. For example, we could use it to build a distributive domain name system, which is simply a mapping between human understandable domain names to IP addresses. We could build a public key directory, which is a mapping between email addresses (or some other form of real-world identity) to public keys.

That's the intuition of what distributed consensus is, but it is useful to provide a technical definition as this will help us determine whether or not a given protocol meets the requirements.

Distributed consensus protocol. There are *n* nodes that each have an input value. Some of these nodes are faulty or malicious. A distributed consensus protocol has the following two properties:

- It must terminate with all honest nodes in agreement on the value
- The value must have been generated by an honest node

What does this mean in the context of Bitcoin? To understand how distributed consensus could work in Bitcoin, remember that Bitcoin is a peer-to-peer system. When Alice wants to pay Bob, what she actually does is broadcast a transaction to all of the Bitcoin nodes that comprise the peer-to-peer network. See Figure 2.1.

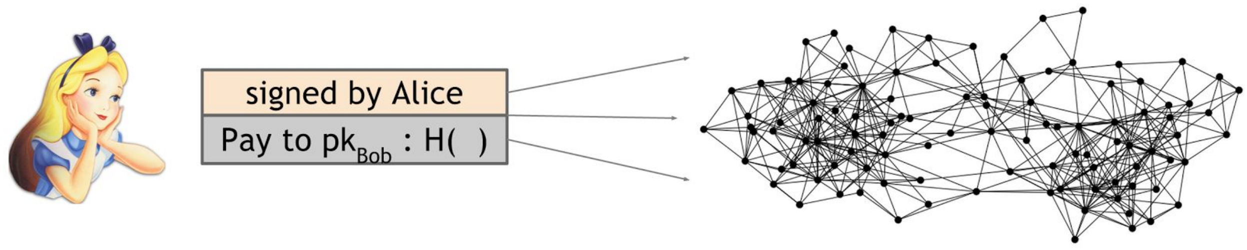


Figure 2.1 Broadcasting a transaction In order to pay Bob, Alice broadcasts the transaction to the entire Bitcoin peer-to-peer network.

Incidentally, you may have noticed that Alice broadcasts the transaction to all the Bitcoin peer-to-peer nodes, but Bob's computer is nowhere in this picture. It's of course possible that Bob is running one of the nodes in the peer-to-peer network. In fact, if he wants to be notified that this transaction did in fact happen and that he got paid, running a node might be a good idea. Nevertheless, there is no requirement that Bob be listening on the network; running a node is not necessary for Bob to receive the funds. The bitcoins will be his whether or not he's operating a node on the network.

What exactly is it that the nodes might want to reach consensus on in the Bitcoin network? Given that a variety of users are broadcasting these transactions to the network, the nodes must agree on exactly which transactions were broadcast and the order in which these transactions happened. This will result in a single, global ledger for the system. Recall that in ScroogeCoin, for optimization, we put transactions into blocks. Similarly, in Bitcoin, we do consensus on a block-by-block basis.

So at any given point, all the nodes in the peer-to-peer network have a ledger consisting of a sequence of blocks, each containing a list of transactions, that they've reached consensus on. Additionally, each node has a pool of outstanding transactions that it has heard about but have not yet been included on the block chain. For these transactions, consensus has not yet happened, and so by definition, each node might have a slightly different version of the outstanding transaction pool. In practice, this occurs because the peer-to-peer network is not perfect, so some nodes may have heard about a transaction that other nodes have not heard about.

How exactly do nodes come to consensus on a block? One way to do this: at regular intervals, say every 10 minutes, every node in the system proposes its own outstanding transaction pool to be the next block. Then the nodes execute some consensus protocol, where each node's input is its own proposed block. Now, some nodes may be malicious and put invalid transactions into their blocks, but we might assume that other nodes will be honest. If the consensus protocol succeeds, a valid block will be selected as the output. Even if the selected block was proposed by only one node, it's a valid output as long as the block is valid. Now there may be some valid outstanding transaction that did not get included in the block, but this is not a problem. If some transaction somehow didn't make it into this particular block, it could just wait and get into the next block.

The approach in the previous paragraph has some similarities to how Bitcoin works, but it's not quite how it works. There are a number of technical problems with this approach. Firstly, consensus in general is a hard problem since nodes might crash or be outright malicious. Secondly, and specifically in the Bitcoin context, the network is highly imperfect. It's a peer-to-peer system, and not all pairs of nodes are connected to each other. There could be faults in the network because of poor Internet connectivity for example, and thus running a consensus protocol in which all nodes must participate is not really possible. Finally, there's a lot of latency in the system because it's distributed all over the Internet.

Sidebar: The Bitcoin protocol must reach consensus in the face of two types of obstacles: imperfections in the network, such as latency and nodes crashing, as well as deliberate attempts by some nodes to subvert the process.

One particular consequence of this high latency is that there is no notion of global time. What this means is that not all nodes can agree to a common ordering of events simply based on observing timestamps. So the consensus protocol cannot contain instructions of the form, "The node that sent the first message in step 1 must do X in step 2." This simply will not work because not all nodes will agree on which message was sent first in the step 1 of the protocol.

Impossibility results. The lack of global time heavily constrains the set of algorithms that can be used in the consensus protocols. In fact, because of these constraints, much of the literature on distributed consensus is somewhat pessimistic, and many impossibility results have been proven. One very well known impossibility result concerns the ***Byzantine Generals Problem***. In this classic problem, the Byzantine army is separated into divisions, each commanded by a general. The generals communicate by messenger in order to devise a joint plan of action. Some generals may be traitors and may intentionally try to subvert the process so that the loyal generals cannot arrive at a unified plan. The goal of this problem is for all of the loyal generals to arrive at the same plan without the traitorous generals being able to cause them to adopt a bad plan. It has been proven that this is impossible to achieve if one-third or more of the generals are traitors.

A much more subtle impossibility result, known for the names of the authors who first proved it, is called the Fischer-Lynch-Paterson impossibility result. Under some conditions, which include the nodes acting in a deterministic manner, they proved that consensus is impossible with even a single faulty process.

Despite these impossibility results, there are some consensus protocols in the literature. One of the better known among these protocols is ***Paxos***. Paxos makes certain compromises. On the one hand, it never produces an inconsistent result. On the other hand, it accepts the trade-off that under certain conditions, albeit rare ones, the protocol can get stuck and fail to make any progress.

Breaking traditional assumptions. But there's good news: these impossibility results were proven in a very specific model. They were intended to study distributed databases, and this model doesn't carry over very well to the Bitcoin setting because Bitcoin violates many of the assumptions built into the models. In a way, the results tell us more about the model than they do about the problem of distributed consensus.

Ironically, with the current state of research, consensus in Bitcoin works better in practice than in theory. That is, we observe consensus working, but have not developed the theory to fully explain why it works. But developing such a theory is important as it can help us predict unforeseen attacks and problems, and only when we have a strong theoretical understanding of how Bitcoin consensus works will we have strong guarantees Bitcoin's security and stability.

What are the assumptions in traditional models for consensus that Bitcoin violates? First, it introduces the idea of incentives, which is novel for a distributed consensus protocol. This is only possible in Bitcoin because it is a currency and therefore has a natural mechanism to incentivize participants to act honestly. So Bitcoin doesn't quite solve the distributed consensus problem in a general sense, but it solves it in the specific context of a currency system.

Second, Bitcoin embraces the notion of randomness. As we will see in the next two sections, Bitcoin's consensus algorithm relies heavily on randomization. Also, it does away with the notion of a specific starting point and ending point for consensus. Instead, consensus happens over a long period of time, about an hour in the practical system. But even at the end of that time, nodes can't be certain that any particular transaction or a block has made it into the ledger. Instead, as time goes on, the probability that your view of any block will match the eventual consensus view increases, and the probability that the views will diverge goes down exponentially. These differences in the model are key to how Bitcoin gets around the traditional impossibility results for distributed consensus protocols.

2.3 Consensus without identity: using a block chain

In this section we'll study the technical details of Bitcoin's consensus algorithm. Recall that Bitcoin nodes do not have persistent, long-term identities. This is another difference from traditional distributed consensus algorithms. One reason for this lack of identities is that in a peer-to-peer system, there is no central authority to assign identities to participants and verify that they're not creating new nodes at will. The technical term for this is a ***Sybil attack***. Sybils are just copies of nodes that a malicious adversary can create to look like there are a lot of different participants, when in fact all those pseudo-participants are really controlled by the same adversary. The other reason is that pseudonymity is inherently a goal of Bitcoin. Even if it were possible or easy to establish identities for all nodes or all participants, we wouldn't necessarily want to do that. Although Bitcoin doesn't give strong anonymity guarantees in that the different transactions that one makes can often be linked together, it does have the property that nobody is forced to reveal their real-life identity, like their name or IP address, in order to participate. And that's an important property and a central feature of Bitcoin's design.

If nodes did have identities, the design would be easier. For starters, identities would allow us to put in the protocol instructions of the form, "Now the node with the lowest numerical ID should take some step." Without identities, the set of possible instructions is more constrained. But a much more serious reason for nodes to have identities is for security. If nodes were identified and it weren't trivial to create new node identities, then we could make assumptions on the number of nodes that are malicious, and we could derive security properties out of that. For both of these reasons, the lack of identities introduces difficulties for the consensus protocol in Bitcoin.

We can compensate for the lack of identities by making a weaker assumption. Suppose there is somehow an ability to pick a random node in the system. A good motivating analogy for this is a lottery or a raffle, or any number of real-life systems where it's hard to track people, give them identities and verify those identities. What we do in those contexts is to give out tokens or tickets or something similar. That enables us to later pick a random token ID, and call upon the owner of that ID. So for the moment, take a leap of faith and assume that it is possible to pick a random node from the Bitcoin network in this manner. Further assume, for the moment, that this token generation and distribution algorithm is sufficiently smart so that if the adversary is going to try to create a lot of Sybil nodes, all of those Sybils together will get only one token. This means the adversary is not able to multiply his power by creating new nodes. If you think this is a lot to assume, don't worry. Later in this chapter, we'll remove these assumptions and show in detail how properties equivalent to these are realized in Bitcoin.

Implicit Consensus. This assumption of random node selection makes possible something called ***implicit consensus***. There are multiple rounds in our protocol, each corresponding to a different block in the block chain. In each round, a random node is somehow selected, and this node gets to propose the next block in the chain. There is no consensus algorithm for selecting the block, and no voting of any kind. The chosen node unilaterally proposes what the next block in the block chain will be. But what if that node is malicious? Well, there is a process for handling that, but it is an implicit one. Other nodes will implicitly accept or reject that block by choosing whether or not to build on top of it. If they accept that block, they will signal their acceptance by extending the block chain including the accepted block. By contrast, if they reject that block, they will extend the chain by ignoring that block, and building on top of whichever is the previous block that they accepted. Recall that each block contains a hash of the block that it extends. This is the technical mechanism that allows nodes to signal which block it is that they are extending.

Bitcoin consensus algorithm (simplified)

This algorithm is simplified in that it assumes the ability to select a random node in a manner that is not vulnerable to Sybil attacks.

1. New transactions are broadcast to all nodes
2. Each node collects new transactions into a block
3. In each round a <u>random</u> node gets to broadcast its block
4. Other nodes accept the block only if all transactions in it are valid (unspent, valid signatures)
5. Nodes express their acceptance of the block by including its hash in the next block they create

Let's now try to understand why this consensus algorithm works. To do this, let's consider how a malicious adversary — who we'll call Alice — may be able to subvert this process.

Stealing Bitcoins. Can Alice simply steal bitcoins belonging to another user at an address she doesn't control? No. Even if it is Alice's turn to propose the next block in the chain, she cannot steal other users' bitcoins. Doing so would require Alice to create a valid transaction that spends that coin. This would require Alice to forge the owners' signatures which she cannot do if a secure digital signature scheme is used. So as long as the underlying cryptography is solid, she's not able to simply steal bitcoins.

Denial of service attack. Let's consider another attack. Say Alice really dislikes some other user Bob. Alice can then decide that she will not include any transactions originating from Bob's address in any block that she proposes to get onto the block chain. In other words, she's denying service to Bob. While this is a valid attack that Alice can try to mount, luckily it's nothing more than a minor annoyance. If Bob's transaction doesn't make it into the next block that Alice proposes, he will just wait until an honest node gets the chance to propose a block and then his transaction will get into that block. So that's not really a good attack either.

Double-spend attack. Alice may try to launch a double-spend attack. To understand how that works, let's assume that Alice is a customer of some online merchant or website run by Bob, who provides some online service in exchange for payment in bitcoins. Let's say Bob's service allows the download of some software. So here's how a double-spend attack might work. Alice adds an item to her shopping cart on Bob's website and the server requests payment. Then Alice creates a Bitcoin transaction from her address to Bob's and broadcasts it to the network. Let's say that some honest node creates the next block, and includes this transaction in that block. So there is now a block that was created by an honest node that contains a transaction that represents a payment from Alice to the merchant Bob.

Recall that a transaction is a data structure that contains Alice's signature, an instruction to pay to Bob's public key, and a hash. This hash represents a pointer to a previous transaction output that Alice received and is now spending. That pointer must reference a transaction that was included in some previous block in the consensus chain.

Note, by the way, that there are two different types of hash pointers here that can easily be confused. Blocks include a hash pointer to the previous block that they're extending. Transactions include one or more hash pointers to previous transaction outputs that are being redeemed.

Let's return to how Alice can launch a double spend attack. The latest block was generated by an honest node and includes a transaction in which Alice pays Bob for the software download. Upon seeing this transaction included in the block chain, Bob concludes that Alice has paid him and allows Alice to download the software. Suppose the next random node that is selected in the next round happens to be controlled by Alice. Now since Alice gets to propose the next block, she could propose a block that ignores the block that contains the payment to Bob and instead contains a pointer to the previous block. Furthermore, in the block that she proposes, Alice includes a transaction that transfers the very coins that she was sending to Bob to a different address that she herself controls. This is a classic double-spend pattern. Since the two transactions spend the same coins, only one of them can be included in the block chain. If Alice succeeds in including the payment to her own address in the block chain, then the transaction in which she pays Bob is useless as it can never be included later in the block chain.

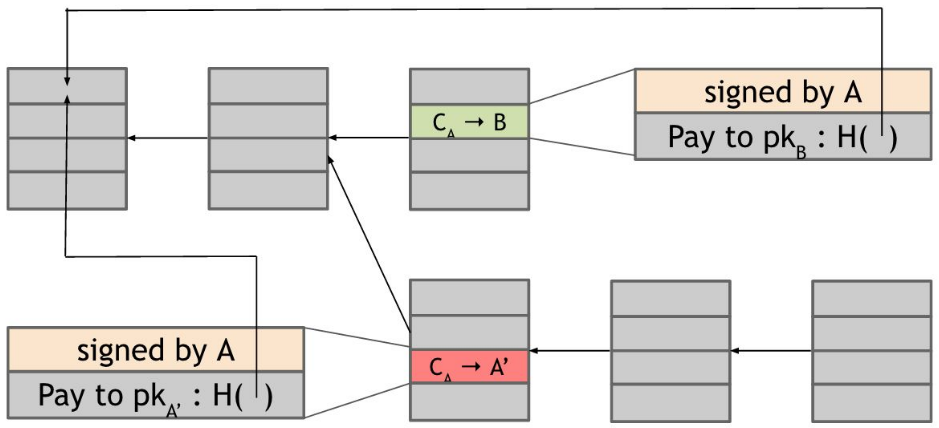


Figure 2.1 A double spend attempt. Alice creates two transactions: one in which she sends Bob Bitcoins, and a second in which she double spends those Bitcoins by sending them to a different address that she controls. As they spend the same Bitcoins, only one of these transactions can be included in the block chain. The arrows are pointers from one block to the previous block that it extends including a hash of that previous block within its own contents. C_A is used to denote a coin owned by Alice.

And how do we know if this double spend attempt is going to succeed or not? Well, that depends on which block will ultimately end up on the long-term consensus chain — the one with the Alice → Bob transaction or the one with the Alice → Alice transaction. What determines which block will be included? Honest nodes follow the policy of extending the longest valid branch, so which branch will they extend? There is no right answer! At this point, the two branches are the same length — they only differ in the last block and both of these blocks are valid. The node that chooses the next block then may decide to build upon either one of them, and this choice will largely determine whether or not the double-spend succeeds.

A subtle point: from a moral point of view, there is a clear difference between the block containing the transaction that pays Bob and the block containing the transaction in which Alice double spends those coins to her own address. But this distinction is only based on our knowledge of the story that Alice first paid Bob and then attempted to double spend. From a technological point of view, however, these two transactions are completely identical and both blocks are equally valid. The nodes that are looking at this really have no way to tell which is the morally legitimate transaction.

In practice, nodes often follow a heuristic of extending the block that they first heard about on the peer-to-peer network. But it's not a solid rule. And in any case, because of network latency, it could easily be that the block that a node first heard about is actually the one that was created second. So there is at least some chance that the next node that gets to propose a block will extend the block containing the double spend. Alice could further try to increase the likelihood of this happening by bribing the next node to do so. If the next node does build on the double-spend block for whatever reason, then this chain will now be longer than the one that includes the transaction to Bob. At this point, the next honest node is much more likely to continue to build on this chain since it is longer. This process will continue, and it will become increasingly likely that the block containing the double-spend will be part of the long-term consensus chain. The block containing the transaction to Bob, on the other hand, gets completely ignored by the network, and this is now called an ***orphan block***.

Let's now reconsider this whole situation from Bob-the-merchant's point of view. Understanding how Bob can protect himself from this double-spending attack is a key part of understanding Bitcoin security. When Alice broadcasts the transaction that represents her payment to Bob, Bob is listening on the network and hears about this transaction even before the next block is created. If Bob was even more foolhardy than we previously described, he can complete the checkout process on the website and allow Alice to download the software right at that moment. That's called a ***zero-confirmation transaction***. This leads to an even more basic double spend attack than the one described before. Previously, for the double-spend attack to occur, we had to assume that a malicious actor controls the node that proposes the next block. But if Bob allows Alice to download the software before the transaction receives even a single confirmation on the block chain, then Alice can immediately broadcast a double-spend transaction, and an honest node may include it in the next block instead of the transaction that pays Bob.

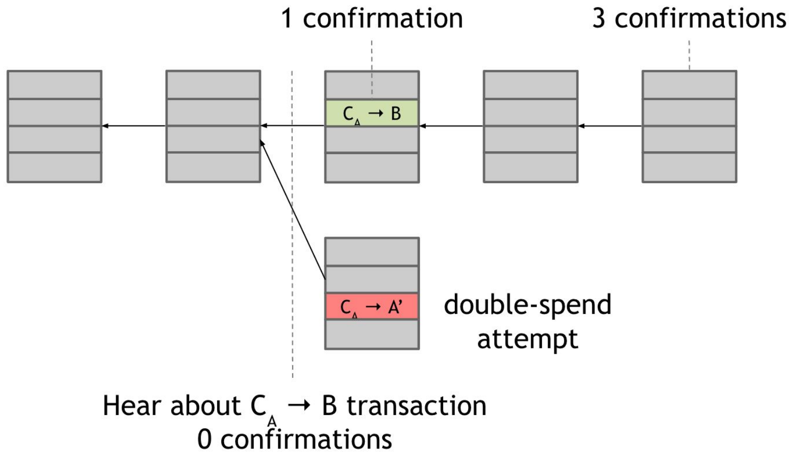


Figure 2.2 Bob the Merchant's view. This is what Alice's double-spend attempt looks like from Bob the merchant's viewpoint. In order to protect himself from this attack, Bob should wait until the transaction with which Alice pays him is included in the block chain and has several confirmations.

On the other hand, a cautious merchant would not release the software to Alice even after the transaction was included in one block, and would continue to wait. If Bob sees that Alice successfully launches a double-spend attack, he realizes that the block containing Alice's payment to him has been orphaned. He should abandon the transaction and not let Alice download the software. Instead, if it happens that despite the double-spend attempt, the next several nodes build on the block with the Alice → Bob transaction, then Bob gains confidence that this transaction will be on the long-term consensus chain.

In general, the more confirmations a transaction gets, the higher the probability that it is going to end up on the long-term consensus chain. Recall that honest nodes' behavior is always to extend the longest valid branch that they see. The chance that the shorter branch with the double spend will catch up to the longer branch becomes increasingly tiny as it grows longer than any other branch. This is especially true if only a minority of the nodes are malicious — for a shorter branch to catch up, several malicious nodes would have to be picked in close succession.

In fact, the double-spend probability decreases exponentially with the number of confirmations. So, if the transaction that you're interested in has received *k* confirmations, then the probability that a double-spend transaction will end up on the long-term consensus chain goes down exponentially as a function of *k*. The most common heuristic that's used in the Bitcoin ecosystem is to wait for six confirmations. There is nothing really special about the number six. It's just a good tradeoff between the amount of time you have to wait and your guarantee that the transaction you're interested in ends up on the consensus block chain.

To recap, protection against invalid transactions is entirely cryptographic. But it is enforced by consensus, which means that if a node does attempt to include a cryptographically invalid transaction, then the only reason that transaction won't end up in the long-term consensus chain is because a majority of the nodes are honest and won't include an invalid transaction in the block chain. On the other hand, protection against double-spending is purely by consensus. Cryptography has nothing to say about this, and two transactions that represent a double-spend attempt are both valid from a cryptographic perspective. But it's the consensus that determines which one will end up on the long-term consensus chain. And finally, you're never 100 percent sure that a transaction you're interested in is on the consensus branch. But, this exponential probability guarantee is rather good. After about six transactions, there's virtually no chance that you're going to go wrong.

2.4 Incentives and proof of work

In the previous section, we got a basic look at Bitcoin's consensus algorithm and a good intuition for why we believe that it's secure. But recall from the beginning of the chapter that Bitcoin's decentralization is partly a technical mechanism and partly clever incentive engineering. So far we've mostly looked at the technical mechanism. Now let's talk about the incentive engineering that happens in Bitcoin.

We asked you to take a leap of faith earlier in assuming that we're able to pick a random node and, perhaps more problematically, that at least 50 percent of the time, this process will pick an honest node. This assumption of honesty is particularly problematic if there are financial incentives for participants to subvert the process, in which case we can't really assume that a node will be honest. The question then becomes: can we give nodes an incentive for behaving honestly?

Consider again the double-spend attempt after one confirmation (Figure 2.2). Can we penalize, somehow, the node that created the block with the double-spend transaction? Well, not really. As we mentioned earlier, it’s hard to know which is the morally legitimate transaction. But even if we did, it’s still hard to punish nodes since they don’t have identities. So instead, let’s flip the question around and ask, can we reward each of the nodes that created the blocks that did end up on the long-term consensus chain? Well, again, since those nodes don’t reveal their real-world identities, we can’t quite mail them cash to their home addresses. If only there were some sort of digital currency that we could use instead... you can probably see where this is going. We’re going to use bitcoins to incentivize the nodes that created these blocks.

Let’s pause for a moment. Everything that we’ve described so far is just an abstract algorithm for achieving distributed consensus and is not specific to the application. Now we’re going to break out of that model, and we’re going to use the fact that the application we’re building through this distributed consensus process is in fact a currency. Specifically, we’re going to incentivize nodes to behave honestly by paying them in units of this currency.

Block Reward. How is this done? There are two separate incentive mechanisms in Bitcoin. The first is the ***block reward***. According to the rules of Bitcoin, the node that creates a block gets to include a special transaction in that block. This transaction is a coin-creation transaction, analogous to CreateCoins in Scroogecoin, and the node can also choose the recipient address of this transaction. Of course that node will typically choose an address belonging to itself. You can think of this as a payment to the node in exchange for the service of creating a block on the consensus chain.

At the time of this writing, the value of the block reward is fixed at 25 Bitcoins. But it actually halves every 210,000 blocks. Based on the rate of block creation that we will see shortly, this means that the rate drops roughly every four years. We’re now in the second period. For the first four years of Bitcoin’s existence, the block reward was 50 bitcoins; now it’s 25. And it’s going to keep halving. This has some interesting consequences, which we will see shortly.

You may be wondering why the block reward incentivizes honest behavior. It may appear, based on what what we’ve said so far, that this node gets the block reward regardless of whether it proposes a valid block or behaves maliciously. But this is not true! Think about it — how will this node “collect” its reward? That will only happen if the block in question ends up on the long-term consensus branch because just like every other transaction, the coin-creation transaction will only be accepted by other nodes if it ends up on the consensus chain. That’s the key idea behind this incentive mechanism. It’s a very subtle but very powerful trick. It incentivizes nodes to behave in whatever way they believe will get other nodes to extend their blocks. So if most of the network is following the longest valid branch rule, it incentivizes all nodes to continue to follow that rule. That’s Bitcoin’s first incentive mechanism.

We mentioned that every 210,000 blocks (or approximately four years), the block reward is cut in half. In figure 2.3, the slope of this curve is going to keep halving. This is a geometric series, and you might know that it means that there is a finite sum. It works out to a total of 21 million bitcoins.

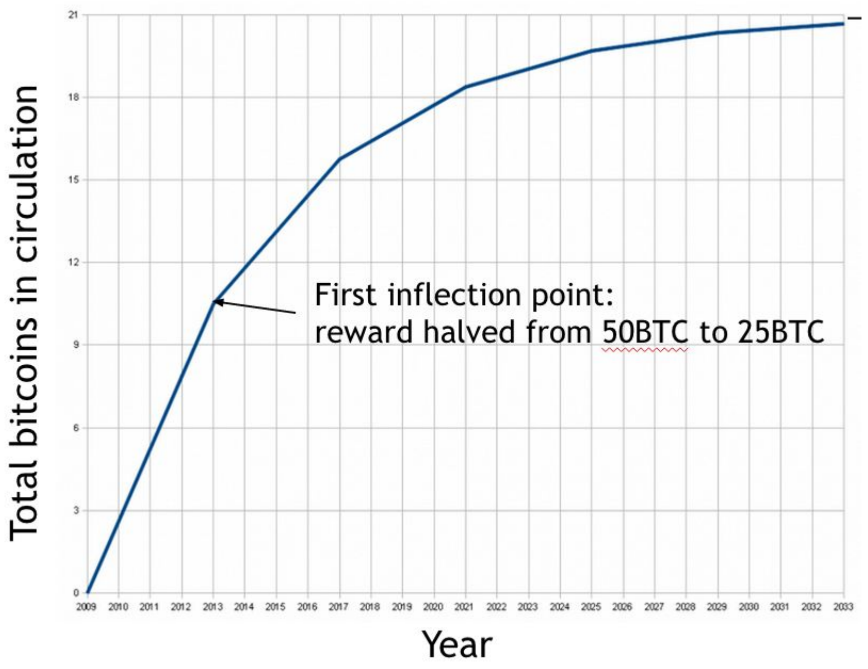


Figure 2.3 The block reward is cut in half every four years limiting the total supply of bitcoins to 21 million.

It is important to note that this is the only way in which new bitcoins are allowed to be created. There is no other coin generation mechanism, and that's why 21 million is a final and total number (as the rules stand now, at least) for how many bitcoins there can ever be. This new block creation reward is actually going to run out in 2140, as things stand now. Does that mean that the system will stop working in 2140 and become insecure because nodes no longer have the incentive to behave honestly? Not quite. The block reward is only the first of two incentive mechanisms in Bitcoin.

Transaction fees The second incentive mechanism is called the ***transaction fee***. The creator of any transaction can choose to make the total value of the transaction outputs less than the total value of its inputs. Whoever creates the block that first puts that transaction into the block chain gets to collect the difference, which acts a transaction fee. So if you're a node that's creating a block that contains, say, 200 transactions, then the sum of all those 200 transaction fees is paid to the address that you put into that block. The transaction fee is purely voluntary, but we expect, based on our understanding of the system, that as the block reward starts to run out, it will become more and more important, almost mandatory, for users to include transaction fees in order to get a reasonable quality of service. To a certain degree, this is already starting to happen now. But it is yet unclear precisely how the system will evolve; it really depends on a lot of game theory which hasn't been fully worked out yet. That's an interesting area of open research in Bitcoin.

There are still a few problems remaining with the consensus mechanism as we described it. The first major one is the leap of faith that we asked you to take that somehow we can pick a random node. Second, we've created a new problem by giving nodes these incentives for participation. The system can become unstable as the incentives cause a free-for-all where everybody wants to run a Bitcoin node in the hope of capturing some of these rewards. And a third one is an even trickier version of this problem, which is that an adversary might create a large number of Sybil nodes to try and subvert the consensus process.

Mining and proof-of-work. It turns out that all of these problems are related, and all of them have the same solution, which is called ***proof-of-work***. The key idea behind proof-of-work is that we approximate the selection of a random node by instead selecting nodes in proportion to a resource that we hope that nobody can monopolize. If, for example, that resource is computing power, then it's a proof-of-work system. Alternately, it could be in proportion to ownership of the currency, and that's called ***proof-of-stake***. Although it's not used in Bitcoin, proof-of-stake is a legitimate alternate model and it's used in other cryptocurrencies. We'll see more about proof-of-stake and other proof-of-work variants in Chapter 8.

But back to proof-of-work. Let's try to get a better idea of what it means to select nodes in proportion to their computing power. Another way of understanding this is that we're allowing nodes to compete with each other by using their computing power, and that will result in nodes automatically being picked in that proportion. Yet another view of proof-of-work is that we're making it moderately hard to create new identities. It's sort of a tax on identity creation and therefore on the Sybil attack. This might all appear a bit vague, so let's go ahead and look at the details of the proof-of-work system that's used in Bitcoin, which should make things a lot clearer.

Bitcoin achieves proof-of-work using ***hash puzzles***. In order to create a block, the node that proposes that block is required to find a number, or ***nonce***, such that when you concatenate the nonce, the previous hash, and the list of transactions that comprise that block and take the hash of this whole string, then that hash output should be a number that falls into a target space that is quite small in relation to the much larger output space of that hash function. We can define such a target space as any value falling below a certain target value. In this case, the nonce will have to satisfy the following inequality:

$$H(nonce \,\|\, prev\_hash \,\|\, tx \,\|\, tx \,\|\, \ldots \,\|\, tx) < target$$

As we saw earlier, normally a block contains a series of transactions that a node is proposing. In addition, a block also contains a hash pointer to the previous block[1]. In addition, we're now requiring that a block also contain a nonce. The idea is that we want to make it moderately difficult to find a nonce that satisfies this required property, which is that hashing the whole block together, including that nonce, is going to result in a particular type of output. If the hash function satisfies the

[1] We are using the term hash pointer loosely. The pointer is just a string in this context as it need not tell us where to find this block. We will find the block by asking other peers on the network for it. The important part is the hash that both acts as an ID when requesting other peers for the block and lets us validate the block once we have obtained it.

puzzle-friendliness property from Chapter 1, then the only way to succeed in solving this hash puzzle is to just try enough nonces one by one until you get lucky. So specifically, if this target space were just one percent of the overall output space, you would have to try about 100 nonces before you got lucky. In reality, the size of this target space is not nearly as high as one percent of the output space. It's much, much smaller than that as we will see shortly.

This notion of hash puzzles and proof of work completely does away with the requirement to magically pick a random node. Instead, nodes are simply independently competing to solve these hash puzzles all the time. Once in a while, one of them will get lucky and will find a random nonce that satisfies this property. That lucky node then gets to propose the next block. That's how the system is completely decentralized. There is nobody deciding which node it is that gets to propose the next block.

Difficult to compute. There are three important properties of hash puzzles. The first is that they need to be quite difficult to compute. We said moderately difficult, but you'll see why this actually varies with time. As of the end of 2014, the difficulty level is about 10^{20} hashes per block. In other words the size of the target space is only $1/10^{20}$ of the size of the output space of the hash function. This is a lot of computation — it's out of the realm of possibility for a commodity laptop, for example. Because of this, only some nodes even bother to compete in this block creation process. This process of repeatedly trying and solving these hash puzzles is known as ***Bitcoin mining***, and we call the participating nodes ***miners***. Even though technically anybody can be a miner, there's been a lot of concentration of power in the mining ecosystem due to the high cost of mining.

Parameterizable cost. The second property is that we want the cost to be parameterizable, not a fixed cost for all time. The way that's accomplished is that all the nodes in the Bitcoin peer-to-peer network will automatically recalculate the target, that is the size of the target space as a fraction of the output space, every 2016 blocks. They recalculate the target in such a way that the average time between successive blocks produced in the Bitcoin network is about 10 minutes. With a 10-minute average time between blocks, 2016 blocks works out to two weeks. In other words, the recalculation of the target happens roughly every two weeks.

Let's think about what this means. If you're a miner, and you've invested a certain fixed amount of hardware into Bitcoin mining, but the overall mining ecosystem is growing, more miners are coming in, or they're deploying faster and faster hardware, that means that over a two week period, slightly more blocks are going to be found than expected. So nodes will automatically readjust the target, and the amount of work that you have to do to be able to find a block is going to increase. So if you put in a fixed amount of hardware investment, the rate at which you find blocks is actually dependent upon what other miners are doing. There's a very nice formula to capture this, which is that the probability that any given miner, Alice, is going to win the next block is equivalent to the fraction of global hash power that she controls. This means that if Alice has mining hardware that's about 0.1 percent of total hash power, she will find roughly one in every 1,000 blocks.

What is the purpose of this readjustment? Why do we want to maintain this 10-minute invariant? The reason is quite simple. If blocks were to come very close together, then there would be a lot of inefficiency, and we would lose the optimization benefits of being able to put a lot of transactions in a single block. There is nothing magical about the number 10, and if you went down from 10 minutes to 5 minutes, it would probably be just fine. There's been a lot of discussion about the ideal block latency that altcoins, or alternative cryptocurrencies, should have. But despite some disagreements about the ideal latency, everybody agrees that it should be a fixed amount. It cannot be allowed to go down without limit. That's why we have the automatic target recalculation feature.

The way that this cost function and proof of work is set up allows us to reformulate our security assumption. Here's where we finally depart from the last leap of faith that we asked you to take earlier. Instead of saying that somehow the majority of nodes are honest in a context where nodes don't even have identities and not being clear about what that means, we can now state crisply, that a lot of attacks on Bitcoin are infeasible if the majority of miners, weighted by hash power, are following the protocol — or, are honest. This is true because if a majority of miners, weighted by hash power, are honest, the competition for proposing the next block will automatically ensure that there is at least a 50 percent chance that the next block to be proposed at any point is coming from an honest node.

Sidebar. in the research fields of distributed systems and computer security, it is common to assume that some percentage of nodes are honest and to show that the system works as intended even if the other nodes behave arbitrarily. That's basically the approach we've taken here, except that we weight nodes by hash power in computing the majority. The original Bitcoin whitepaper contains this type of analysis as well.

But the field of game theory provides an entirely different, and arguably more sophisticated and realistic way to determine how a system will behave. In this view, we don't split nodes into honest and malicious. Instead, we assume that *every* node acts according to its incentives. Each node picks a (randomized) strategy to maximize its payoff, taking into account other nodes' potential strategies. If the protocol and incentives are designed well, then most nodes will follow the rules most of the time. "Honest" behavior is just one strategy of many, and we attach no particular moral salience to it.

In the game theoretic view, the big question is whether the default miner behavior is a "Nash equilibrium," that is, whether it represents a stable situation in which no miner can realize a higher payoff by deviating from honest behavior. This question is still contentious and an active area of research.

Solving hash puzzles is probabilistic because nobody can predict which nonce is going to result in solving the hash puzzle. The only way to do it is to try nonces one by one and hope that one succeeds. Mathematically, this process is called **Bernoulli trials**. A Bernoulli trial is an experiment with two possible outcomes, and the probability of each outcome occurring is fixed between successive trials. Here, the two outcomes are whether or not the hash falls in the target, and assuming the hash functions behaves like a random function, the probability of those outcomes is fixed. Typically, nodes try so many nonces that Bernoulli trials, a discrete probability process, can be well approximated by a continuous probability process called a **Poisson process**, a process in which events occur independently at a constant average rate. The end result of all of that is that the probability density function that shows the relative likelihood of the time until the next block is found looks like Figure 2.4.

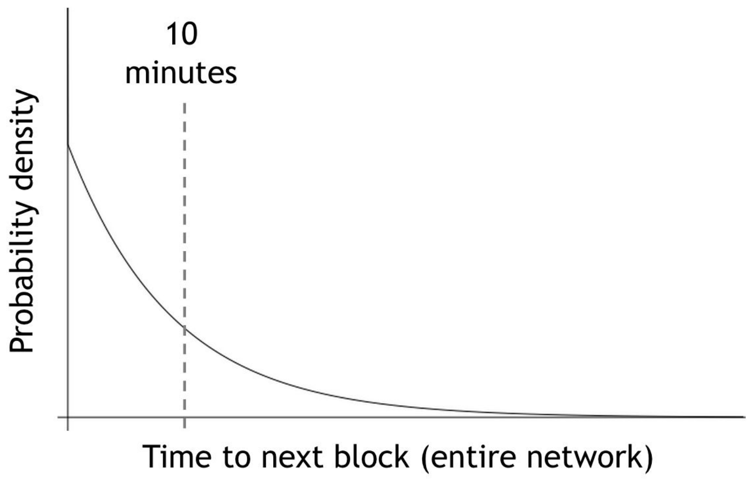


Figure 2.4 Probability density function of the time until the next block is found.

This is known as an exponential distribution. There is some small probability that if a block has been found now, the next block is going to be found very soon, say within a few seconds or a minute. And there is also some small probability that it will take a long time, say an hour, to find the next block. But overall, the network automatically adjusts the difficulty so that the inter-block time is maintained at an average, long term, of 10 minutes. Notice that Figure 2.3 shows how frequently blocks are going to be created by the entire network not caring about which miner actually finds the block.

If you're a miner, you're probably interested in how long it will take you to find a block. What does this probability density function look like? It's going to have the same shape, but it's just going to have a different scale on the x-axis. Again, it can be represented by a nice equation.
For a specific miner:

$$\textit{mean time to next block} = \frac{10 \textit{ minutes}}{\textit{fraction of hash power}}$$

If you have 0.1 percent of the total network hash power, this equation tells us that you're going to find blocks once every 10,000 minutes, which is just about a week. Not only is your mean time between blocks going to be very high, but the variance of the time between blocks found by you is also going to be very high. This has some important consequences that we're going to look at in chapter 5.

Trivial to verify. Let's now turn to the third important property of this proof of work function, which is that it is trivial to verify that a node has computed proof of work correctly. Even if it takes a node, on average, 10^{20} tries to find a nonce that makes the block hash fall below the target, that nonce must be published as part of the block. It is thus trivial for any other node to look at the block contents, hash them all together, and verify that the output is less than the target. This is quite an important property because, once again, it allows us to get rid of centralization. We don't need any centralized authority verifying that miners are doing their job correctly. Any node or any miner can instantly verify that a block found by another miner satisfies this proof-of-work property.

2.5: Putting it all together

Cost of mining. Let's now look at mining economics. We mentioned it's quite expensive to operate as a miner. At the current difficulty level, finding a single block takes computing about 10^{20} hashes and the block reward is about 25 Bitcoins, which is a sizable amount of money at the current exchange rate. These numbers allow for an easy calculation of whether it's profitable for one to mine, and we can capture this decision with a simple statement:

```
If
    mining reward > mining cost
    then miner profits
where
    mining reward = block reward + tx fees
    mining cost = hardware cost + operating costs (electricity, cooling, etc.)
```

Fundamentally, the mining reward that the miner gets is in terms of the block reward and transaction fees. The miner asks himself how it compares to the total expenditure, which is the hardware and electricity cost.

But there are some complications to this simple equation. The first is that, as you may have noticed, the hardware cost is a fixed cost whereas the electricity cost is a variable cost that is incurred over time. Another complication is that the reward that miners get depends upon the rate at which they find blocks, which depends on not just the power of their hardware, but on the ratio of their hash rate to the total global hash rate. A third complication is that the costs that the miner incurs are typically denominated in dollars or some other traditional currency, but their reward is denominated in bitcoin. So this equation has a hidden dependence on Bitcoin's exchange rate at any given time. And finally, so far we've assumed that the miner is interested in honestly following the protocol. But the miner might choose to use some other mining strategy instead of always attempting to extend the longest valid branch. So this equation doesn't capture all the nuances of the different strategies that the miner can employ. Actually analyzing whether it makes sense to mine is a complicated game theory problem that's not easily answered.

At this point, we've obtained a pretty good understanding of how a Bitcoin achieves decentralization. We will now recap the high level points and put it all together in order to get an even better understanding.

Let's start from identities. As we've learned, there are no real-world identities required to participate in the Bitcoin protocol. Any user can create a pseudonymous key pair at any moment, any number of them. When Alice wants to pay Bob in bitcoins, the Bitcoin protocol does not specify how Alice learns Bob's address. Given these pseudonymous key pairs as identities, transactions are basically messages that are broadcast to the Bitcoin peer-to-peer network that are instructions to transfer coins from one address to another. Bitcoins are just transaction outputs, and we will discuss this in much more detail in the next chapter.

Sidebar. Bitcoin doesn't have fixed denominations like US dollars, and in particular, there is no special designation of "1 bitcoin." Bitcoins are just transaction outputs, and in the current rules, they can have an arbitrary value with 8 decimal places of precision. The smallest possible value is 0.00000001 BTC (bitcoins), which is called 1 **Satoshi**.

The goal of the Bitcoin peer-to-peer network is to propagate all new transactions and new blocks to all the Bitcoin peer nodes. But the network is highly imperfect, and does a best-effort attempt to relay this information. The security of the system doesn't come from the perfection of the peer-to-peer network. Instead, the security comes from the block chain and the consensus protocol that we devoted much of this chapter to studying.

When we say that a transaction is included in the block chain, what we really mean is that the transaction has achieved numerous confirmations. There's no fixed number to how many confirmations are necessary before we are sufficiently convinced of its inclusion, but six is a commonly-used heuristic. The more confirmations a transaction has received, the more certain you can be that this transaction is part of the consensus chain. There will often be orphan blocks, or blocks that don't make it into the consensus chain. There are a variety of reasons that could lead to a block being orphaned. The block may contain an invalid transaction, or a double-spend attempt. It could also just be a result of network latency. That is, two miners may simply end up finding new blocks within just a few seconds of each other. So both of these blocks were broadcast nearly simultaneously onto the network, and one of them will inevitably be orphaned.

Finally, we looked at hash puzzles and mining. Miners are special types of nodes that decide to compete in this game of creating new blocks. They're rewarded for their effort in terms of both newly minted bitcoins (the new-block reward) and existing bitcoins (transaction fees), provided that other miners build upon their blocks. A subtle but crucial point: say that Alice and Bob are two different miners, and Alice has 100 times as much computing power as Bob. This does not mean that Alice will always win the race against Bob to find the next block. Instead, Alice and Bob have a probability ratio of finding the next block, in the proportion 100 to 1. In the long term, Bob will find, on average, one percent of the number of blocks that Alice finds.

We expect that miners will typically be somewhere close to the economic equilibrium in the sense that the expenditure that they incur in terms of hardware and electricity will be roughly equal to the rewards that they obtain. The reason is that if a miner is consistently making a loss, she will probably stop mining. On the other hand, and if mining is very profitable given typical hardware and electricity costs, then more mining hardware would enter the network. The increased hash rate would lead to an increase in the difficulty, and each miner's expected reward would drop.

This notion of distributed consensus permeates Bitcoin quite deeply. In a traditional currency, consensus does come into play to a certain limited extent. Specifically, there is a consensus process that determines the exchange rate of the currency. That is certainly true in Bitcoin as well; We need consensus around the value of Bitcoin. But in Bitcoin, additionally, we need consensus on the state of the ledger, which is what the block chain accomplishes. In other words, even the accounting of how many bitcoins you own is subject to consensus. When we say that Alice owns a certain amount or number of bitcoins, what we actually mean is that the Bitcoin peer-to-peer network, as recorded in the block chain, considers the sum total of all Alice's addresses to own that number of bitcoins. That is ultimate nature of truth in Bitcoin: ownership of bitcoins is nothing more than other nodes agreeing that a given party owns those bitcoins.

Finally, we need consensus about the rules of the system because occasionally, the rules of the system have to change. There are two types of changes to the rules of Bitcoin, known respectively as ***soft forks*** and ***hard forks***. We're going to defer this discussion of the differences to later chapters in which we will discuss them in detail.

Getting a cryptocurrency off the ground. Another subtle concept is that of ***bootstrapping***. There is a tricky interplay between three different ideas in Bitcoin: the security of the block chain, the health of the mining ecosystem, and the value of the currency. We obviously want the block chain to be secure for Bitcoin to be a viable currency. For the block chain to be secure, an adversary must not be able to overwhelm the consensus process. This in turn means that an adversary cannot create a lot of mining nodes and take over 50 percent or more of the new block creation.

But when will that be true? A prerequisite is having a healthy mining ecosystem made up of largely honest, protocol-following nodes. But what's a prerequisite for that — when can we be sure that a lot of miners will put a lot of computing power into participating in this hash puzzle solving competition? Well, they're only going to do that if the exchange rate of Bitcoin is pretty high because the rewards that they receive are denominated in Bitcoins whereas their expenditure is in dollars. So the more the value of the currency goes up, the more incentivized these miners are going to be.

But what ensures a high and stable value of the currency? That can only happen if users in general have trust in the security of the block chain. If they believe that the network could be overwhelmed at any moment by an attacker, then Bitcoin is not going to have a lot of value as a currency. So you have this interlocking interdependence between the security of the block chain, a healthy mining ecosystem and the exchange rate.

Because of the cyclical nature of this three-way dependence, the existence of each of these is predicated on the existence of the others. When Bitcoin was first created, none of these three existed. There were no miners other than Nakamoto himself running the mining software. Bitcoin didn't have a lot of value as a currency. And the block chain was, in fact, insecure because there was not a lot of mining going on and anybody could have easily overwhelmed this process.

There's no simple explanation for how Bitcoin went from not having any of these properties to having all three of them. Media attention was part of the story — the more people hear about Bitcoin, the more they're going to get interested in mining. And the more they get interested in mining, the more confidence people will have in the security of the block chain because there's now more mining activity going on, and so forth. Incidentally, every new Altcoin that wants to succeed also has to somehow solve this problem of pulling itself up by its bootstraps.

51-percent attack. Finally, let's consider what would happen if consensus failed and there was in fact a ***51-percent attacker*** who controls 51 percent or more of the mining power in the Bitcoin network. We'll consider a variety of possible attacks and see which ones can actually be carried out by such an attacker.

First of all, can this attacker steal coins from an existing address? As you may have guessed, the answer is no, because stealing from an existing address is not possible unless you subvert the cryptography. It's not enough to subvert the consensus process. This is not completely obvious. Let's say the 51 percent attacker creates an invalid block that contains an invalid transaction that represents stealing Bitcoins from an existing address that the attacker doesn't control and transferring them to his own address. The attacker can pretend that it's a valid transaction and keep building upon this block. The attacker can even succeed in making that the longest branch. But the other honest nodes are simply not going to accept this block with an invalid transaction and are going to keep mining based on the last valid block that they found in the network. So what will happen is that there will be what we call a fork in the chain.

Now imagine this from the point of view of the attacker trying to spend these invalid coins, and send them to some merchant Bob as payment for some goods or service. Bob is presumably running a Bitcoin node himself, and it will be an honest node. Bob's node will reject that branch as invalid because it contains an invalid transaction. It's invalid because the signatures didn't check out. So Bob's node will simply ignore the longest branch because it's an invalid branch. And because of that, subverting consensus is not enough. You have to subvert cryptography to steal bitcoins. So we conclude that this attack is not possible for a 51 percent attacker.

We should note that all this is only a thought experiment. If there were, in fact, actual signs of a 51 percent attack, what will probably happen is that the developers will notice this and react to it. They will update the Bitcoin software, and we might expect that the rules of the system, including the peer-to-peer network, might change in some form to make it more difficult for this attack to succeed.

But we can't quite predict that. So we're working in a simplified model where a 51 percent attack happens, but other than that, there are no changes or tweaks to the rules of the system.

Let's consider another attack. Can the 51-percent attacker suppress some transactions? Let's say there is some user, Carol, whom the attacker really doesn't like. The attacker knows some of Carol's addresses, and wants to make sure that no coins belonging to any of those addresses can possibly be spent. Is that possible? Since he controls the consensus process of the block chain, the attacker can simply refuse to create any new blocks that contain transactions from one of Carol's addresses. The attacker can further refuse to build upon blocks that contain such transactions. However, he can't prevent these transactions from being broadcast to the peer-to-peer network because the network doesn't depend on the block chain, or on consensus, and we're assuming that the attacker doesn't fully control the network. The attacker cannot stop the transactions from reaching the majority of nodes, so even if the attack succeeds, it will at least be apparent that the attack is happening.

Can the attacker change the block reward? That is, can the attacker start pretending that the block reward is, instead of 25 Bitcoins, say 100 Bitcoins? This is a change to the rules of the system, and because the attacker doesn't control the copies of the Bitcoin software that all of the honest nodes are running, this is also not possible. This is similar to the reason why the attacker cannot include invalid transactions. Other nodes will simply not recognize the increase in the block reward, and the attacker will thus be unable to spend them.

Finally, can the attacker somehow destroy confidence in Bitcoin? Well, let's imagine what would happen. If there were a variety of double-spend attempts, situations in which nodes did not extend the longest valid branch, and other attempted attacks, then people are going to likely decide that Bitcoin is no longer acting as a decentralized ledger that they can trust. People will lose confidence in the currency, and we might expect that the exchange rate of Bitcoin will plummet. In fact, if it is known that there is a party that controls 51 percent of the hash power, then it's possible that people will lose confidence in Bitcoin even if the attacker is not necessarily trying to launch any attacks. So it is not only possible, but in fact likely, that a 51 percent attacker of any sort will destroy confidence in the currency. Indeed, this is the main practical threat if a 51 percent attack were ever to materialize. Considering the amount of expenditure that the adversary would have to put into attacking Bitcoin and achieving a 51 percent majority, none of the other attacks that we described really make sense from a financial point of view.

Hopefully, at this point you've obtained a really good understanding of how decentralization is achieved in Bitcoin. You should have a good command on how identities work in Bitcoin, how transactions are propagated and validated, the role of the peer-to-peer network in Bitcoin, how the block chain is used to achieve consensus, and how hash puzzles and mining work. These concepts provide a solid foundation and a good launching point for understanding a lot of the more subtle details and nuances of Bitcoin, which we're going to see in the coming chapters.

Further reading

The Bitcoin whitepaper:

Nakamoto, Satoshi. ***Bitcoin: A peer-to-peer electronic cash system*****. (2008)**

The original application of proof-of-work:

Back, Adam. ***Hashcash-a denial of service counter-measure.*** **(2002)**

The Paxos algorithm for consensus:

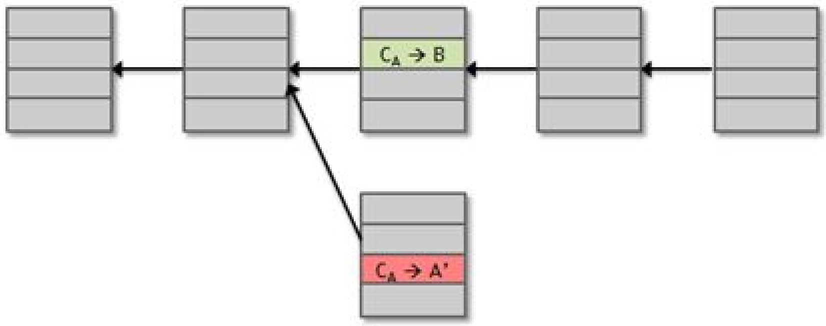
Lamport, Leslie. ***Paxos made simple.*** **ACM Sigact News 32.4 (2001): 18-25.**

Exercises

1. Why do miners run "full nodes" that keep track of the entire block chain[2] whereas Bob the merchant can get away with a "lite node" that implements "simplified payment verification," needing to examine only the last few blocks?

2. If a malicious ISP completely controls a user's connections, can it launch a double-spend attack against the user? How much computational effort would this take?

3. Consider Bob the merchant deciding whether or not to accept the $C_A \rightarrow B$ transaction. What Bob is really interested in is whether or not the other chain will catch up. Why, then, does he simply check how many confirmations $C_A \rightarrow B$ has received, instead of computing the difference in length between the two chains?



[2] This only applies to "solo" miners who're not part of a mining pool, but we haven't discussed that yet.

4. Even when all nodes are honest, blocks will occasionally get orphaned: if two miners Minnie and Mynie discover blocks nearly simultaneously, neither will have time to hear about the other's block before broadcasting hers.

4a. What determines whose block will end up on the consensus branch?

4b. What factors affect the rate of orphan blocks? Can you derive a formula for the rate based on these parameters?

4c. Try to empirically measure this rate on the Bitcoin network.

4d. If Mynie hears about Minnie's block just before she's about to discover hers, does that mean she wasted her effort?

4e. Do all miners have their blocks orphaned at the same rate, or are some miners affected disproportionately?

5a. How can a miner establish an identity in a way that's hard to fake? (i.e., anyone can tell which blocks were mined by her.)

5b. If a miner misbehaves, can other miners "boycott" her by refusing to build on her blocks on an ongoing basis?

6a. Assuming that the total hash power of the network stays constant, what is the probability that a block will be found in the next 10 minutes?

6b. Suppose Bob the merchant wants to have a policy that orders will ship within *x* minutes after receipt of payment. What value of *x* should Bob choose so that with 99% confidence 6 blocks will be found within *x* minutes?

Bitcoin and Cryptocurrency Technologies

**Arvind Narayanan, Joseph Bonneau, Edward Felten,
Andrew Miller, Steven Goldfeder**

Draft — Oct 6, 2015

Feedback welcome! Email bitcoinbook@lists.cs.princeton.edu

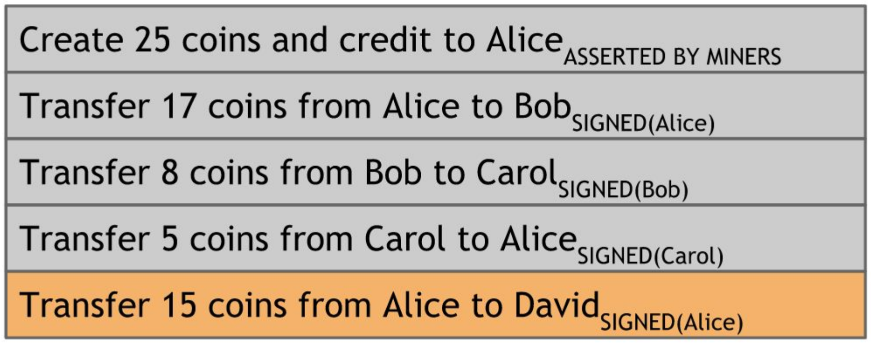
Chapter 3: Mechanics of Bitcoin

This chapter is about the mechanics of Bitcoin. Whereas in the first two chapters, we've talked at a relatively high level, now we're going to delve into detail. We'll look at real data structures, real scripts, and try to learn the details and language of Bitcoin in a precise way to set up everything that we want to talk about in the rest of this book. This chapter will be challenging because a lot of details will be flying at you. You'll learn the specifics and the quirks that make Bitcoin what it is.

To recap where we left off last time, the Bitcoin consensus mechanism gives us an append-only ledger, a data structure that we can only write to. Once data is written to it, it's there forever. There's a decentralized protocol for establishing consensus about the value of that ledger, and there are miners who perform that protocol and validate transactions. Together they make sure that transactions are well formed, that they aren't already spent, and that the ledger and network can function as a currency. At the same time, we assumed that a currency existed to motivate these miners. In this chapter we'll look at the details of how we actually build that currency, to motivate the miners that make this whole process happen.

3.1 Bitcoin transactions

Let's start with transactions, Bitcoin's fundamental building block. We're going to use a simplified model of a ledger for the moment. Instead of blocks, let's suppose individual transactions are added to the ledger one at a time.



Create 25 coins and credit to Alice$_{\text{ASSERTED BY MINERS}}$

Transfer 17 coins from Alice to Bob$_{\text{SIGNED(Alice)}}$

Transfer 8 coins from Bob to Carol$_{\text{SIGNED(Bob)}}$

Transfer 5 coins from Carol to Alice$_{\text{SIGNED(Carol)}}$

Transfer 15 coins from Alice to David$_{\text{SIGNED(Alice)}}$

Figure 3.1 an account-based ledger

How can we build a currency on top of such a ledger? The first model you might think of, which is actually the mental model many people have for how Bitcoin works, is that you have an account-based system. You can add some transactions that create new coins and credit them to somebody. And then later you can transfer them. A transaction would say something like "we're moving 17 coins from Alice to Bob", and it will be signed by Alice. That's all the information about the transaction that's contained in the ledger. In Figure 3.1, after Alice receives 25 coins in the first transaction and then transfers 17 coins to Bob in the second, she'd have 8 Bitcoins left in her account.

The downside to this way of doing things is that anyone who wants to determine if a transaction is valid will have to keep track of these account balances. Take another look at Figure 3.1. Does Alice have the 15 coins that she's trying to transfer to David? To figure this out, you'd have to look backwards in time forever to see every transaction affecting Alice, and whether or not her net balance at the time that she tries to transfer 15 coins to David is greater than 15 coins. Of course we can make this a little bit more efficient with some data structures that track Alice's balance after each transaction. But that's going to require a lot of extra housekeeping besides the ledger itself.

Because of these downsides, Bitcoin doesn't use an account-based model. Instead, Bitcoin uses a ledger that just keeps track of transactions similar to ScroogeCoin in Chapter 1.

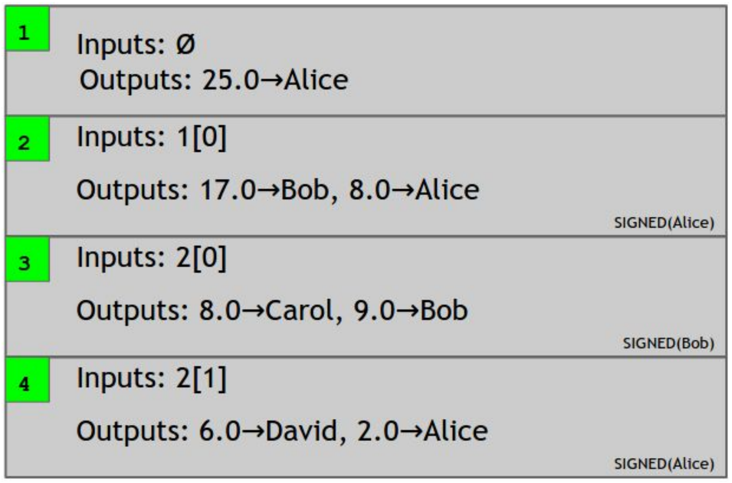


Figure 3.2 a transaction-based ledger, which is very close to Bitcoin

Transactions specify a number of inputs and a number of outputs (recall PayCoins in ScroogeCoin). You can think of the inputs as coins being consumed (created in a previous transaction) and the outputs as coins being created. For transactions in which new currency is being minted, there are no coins being consumed (recall CreateCoins in ScroogeCoin). Each transaction has a unique identifier. Outputs are indexed beginning with 0, so we will refer to the first output as "output 0".

Let's now work our way through Figure 3.2. Transaction 1 has no inputs because this transaction is creating new coins, and it has an output of 25 coins going to Alice. Also, since this is a transaction where new coins are being created, no signature is required. Now let's say that Alice wants to send some of those coins over to Bob. To do so, she creates a new transaction, transaction 2 in our example. In the transaction, she has to explicitly refer to the previous transaction where these coins are coming from. Here, she refers to output 0 of transaction 1 (indeed the only output of transaction 1), which assigned 25 bitcoins to Alice. She also must specify the output addresses in the transaction.

In this example, Alice specifies two outputs, 17 coins to Bob, and 8 coins to Alice. And, of course, this whole thing is signed by Alice, so that we know that Alice actually authorizes this transaction.

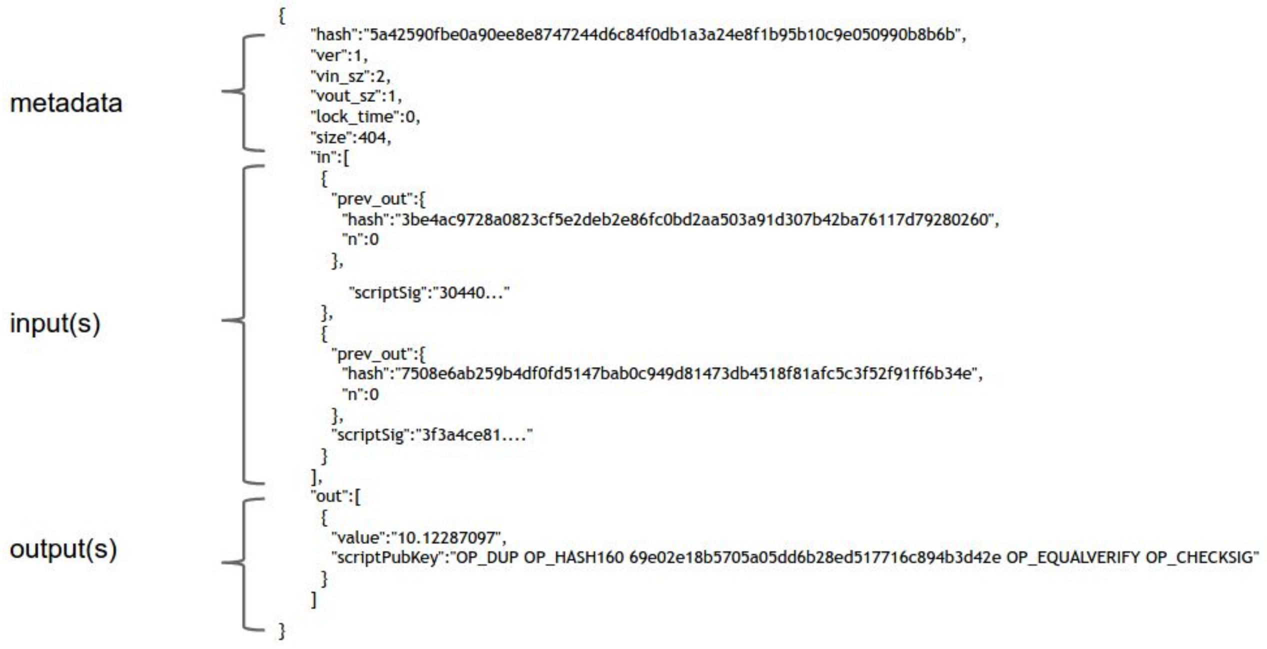
Change addresses. Why does Alice have to send money to herself in this example? Just as coins in ScroogeCoin are immutable, in Bitcoin, the entirety of a transaction output must be consumed by another transaction, or none of it. Alice only wants to pay 17 bitcoins to Bob, but the output that she owns is worth 25 bitcoins. So she needs to create a new output where 8 bitcoins are sent back to herself. It could be a different address from the one that owned the 25 bitcoins, but it would have to be owned by her. This is called a ***change address***.

Efficient verification. When a new transaction is added to the ledger, how easy is it to check if it is valid? In this example, we need to look up the transaction output that Alice referenced, make sure that it has a value of 25 bitcoins, and that it hasn't already been spent. Looking up the transaction output is easy since we're using hash pointers. To ensure it hasn't been spent, we need to scan the block chain between the referenced transaction and the latest block. We don't need to go all the way back to the beginning of the block chain, and it doesn't require keeping any additional data structures (although, as we'll see, additional data structures will speed things up).

Consolidating funds. As in ScroogeCoin, since transactions can have many inputs and many outputs, splitting and merging value is easy. For example, say Bob received money in two different transactions — 17 bitcoins in one, and 2 in another. Bob might say, I'd like to have one transaction I can spend later where I have all 19 bitcoins. That's easy — he creates a transaction with the two inputs and one output, with the output address being one that he owns. That lets him consolidate those two transactions.

Joint payments. Similarly, joint payments are also easy to do. Say Carol and Bob both want to pay David. They can create a transaction with two inputs and one output, but with the two inputs owned by two different people. And the only difference from the previous example is that since the two outputs from prior transactions that are being claimed here are from different addresses, the transaction will need two separate signatures — one by Carol and one by Bob.

Transaction syntax. Conceptually that's really all there is to a Bitcoin transaction. Now let's see how it's represented at a low level in Bitcoin. Ultimately, every data structure that's sent on the network is a string of bits. What's shown in Figure 3.3 is very low-level, but this further gets compiled down to a compact binary format that's not human-readable.



```
{
    "hash":"5a42590fbe0a90ee8e8747244d6c84f0db1a3a24e8f1b95b10c9e050990b8b6b",
    "ver":1,
    "vin_sz":2,
    "vout_sz":1,
    "lock_time":0,
    "size":404,
    "in":[
      {
        "prev_out":{
          "hash":"3be4ac9728a0823cf5e2deb2e86fc0bd2aa503a91d307b42ba76117d79280260",
          "n":0
        },
           "scriptSig":"30440..."
      },
      {
        "prev_out":{
          "hash":"7508e6ab259b4df0fd5147bab0c949d81473db4518f81afc5c3f52f91ff6b34e",
          "n":0
        },
        "scriptSig":"3f3a4ce81...."
      }
    ],
    "out":[
      {
        "value":"10.12287097",
        "scriptPubKey":"OP_DUP OP_HASH160 69e02e18b5705a05dd6b28ed517716c894b3d42e OP_EQUALVERIFY OP_CHECKSIG"
      }
    ]
}
```

Figure 3.3 **An actual Bitcoin transaction.**

As you can see in Figure 3.3, there are three parts to a transaction: some metadata, a series of inputs, and a series of outputs.

- ***Metadata***. There's some housekeeping information — the size of the transaction, the number of inputs, and the number of outputs. There's the hash of the entire transaction which serves as a unique ID for the transaction. That's what allows us to use hash pointers to reference transactions. Finally there's a "lock_time" field, which we'll come back to later.
- ***Inputs.*** The transaction inputs form an array, and each input has the same form. An input specifies a previous transaction, so it contains a hash of that transaction, which acts as a hash pointer to it. The input also contains the index of the previous transaction's outputs that's being claimed. And then there's a signature. Remember that we have to sign to show that we actually have the ability to claim those previous transaction outputs.
- ***Outputs.*** The outputs are again an array. Each output has just two fields. They each have a value, and the sum of all the output values has to be less than or equal to the sum of all the input values. If the sum of the output values is less than the sum of the input values, the difference is a transaction fee to the miner who publishes this transaction.

  And then there's a funny line that looks like what we want to be the recipient address. Each output is supposed to go to a specific public key, and indeed there is something in that field that looks like it's the hash of a public key. But there's also some other stuff that looks like a set of commands. Indeed, this field is a script, and we'll discuss this presently.

3.2 Bitcoin Scripts

Each transaction output doesn't just specify a public key. It actually specifies a script. What is a script, and why do we use scripts? In this section we'll study the Bitcoin scripting language and understand why a script is used instead of simply assigning a public key.

The most common type of transaction in Bitcoin is to redeem a previous transaction output by signing with the correct key. In this case, we want the transaction output to say, "this can be redeemed by a signature from the owner of address X." Recall that an address is a hash of a public key. So merely specifying the address X doesn't tell us what the public key is, and doesn't give us a way to check the signature! So instead the transaction output must say: "this can be redeemed by a public key that hashes to X, along with a signature from the owner of that public key." As we'll see, this is exactly what the most common type of script in Bitcoin says.

```
OP_DUP
OP_HASH160
69e02e18...
OP_EQUALVERIFY
OP_CHECKSIG
```

Figure 3.4. an example Pay-to-PubkeyHash script, the most common type of output script in Bitcoin

But what happens to this script? Who runs it, and how exactly does this sequence of instructions enforce the above statement? The secret is that the inputs also contain scripts instead of signatures. To validate that a transaction redeems a previous transaction output correctly, we combine the new transaction's input script and the earlier transaction's output script. We simply concatenate them, and the resulting script must run successfully in order for the transaction to be valid. These two scripts are called ***scriptPubKey*** and ***scriptSig*** because in the simplest case, the output script just specifies a public key (or an address to which the public key hashes), and the input script specifies a signature with that public key. The combined script can be seen in Figure 3.5.

Bitcoin scripting language. The scripting language was built specifically for Bitcoin, and is just called 'Script' or the Bitcoin scripting language. It has many similarities to a language called Forth, which is an old, simple, stack-based, programming language. But you don't need to understand Forth to understand Bitcoin scripting. The key design goals for Script were to have something simple and compact, yet with native support for cryptographic operations. So, for example, there are special-purpose instructions to compute hash functions and to compute and verify signatures.

The scripting language is stack-based. This means that every instruction is executed exactly once, in a linear manner. In particular, there are no loops in the Bitcoin scripting language. So the number of instructions in the script gives us an upper bound on how long it might take to run and how much memory it could use. The language is not Turing-complete, which means that it doesn't have the ability to compute arbitrarily powerful functions. And this is by design — miners have to run these scripts, which are submitted by arbitrary participants in the network. We don't want to give them the power to submit a script that might have an infinite loop.

```
<sig>
<pubKey>
--------------
OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG
```

Figure 3.5. To check if a transaction correctly redeems an output, we create a combined script by appending the scriptPubKey of the referenced output transaction (bottom) to the scriptSig of the redeeming transaction (top). Notice that <pubKeyHash?> contains a '?'. We use this notation to indicate that we will later check to confirm that this is equal to the hash of the public key provided in the redeeming script.

There are only two possible outcomes when a Bitcoin script is executed. It either executes successfully with no errors, in which case the transaction is valid. Or, if there's any error while the script is executing, the whole transaction will be invalid and shouldn't be accepted into the block chain.

The Bitcoin scripting language is very small. There's only room for 256 instructions, because each one is represented by one byte. Of those 256, 15 are currently disabled, and 75 are reserved. The reserved instruction codes haven't been assigned any specific meaning yet, but might be instructions that are added later in time.

Many of the basic instructions are those you'd expect to be in any programming language. There's basic arithmetic, basic logic — like 'if' and 'then' — , throwing errors, not throwing errors, and returning early. Finally, there are crypto instructions which include hash functions, instructions for signature verification, as well as a special and important instruction called CHECKMULTISIG that lets you check multiple signatures with one instruction. Figure 3.6 lists some of the most common instructions in the Bitcoin scripting language.

The CHECKMULTISIG instruction requires specifying *n* public keys, and a parameter *t*, for a threshold. For this instruction to execute validly, there have to be at least *t* signatures from *t* out of *n* of those public keys that are valid. We'll show some examples of what you'd use multisignatures for in the next section, but it should be immediately clear this is quite a powerful primitive. We can express in a compact way the concept that *t* out of *n* specified public keys must sign in order for the transaction to be valid.

Incidentally, there's a bug in the multisignature implementation, and it's been there all along. The CHECKMULTISIG instruction pops an extra data value off the stack and ignores it. This is just a quirk of the Bitcoin language and one has to deal with it by putting an extra dummy variable onto the stack. The bug was in the original implementation, and the costs of fixing it are much higher than the damage it causes, as we'll see later in Section 3.5. At this point, this bug is considered a feature in Bitcoin, in that it's not going away.

`OP_DUP`	Duplicates the top item on the stack
`OP_HASH160`	Hashes twice: first using SHA-256 and then RIPEMD-160
`OP_EQUALVERIFY`	Returns true if the inputs are equal. Returns false and marks the transaction as invalid if they are unequal
`OP_CHECKSIG`	Checks that the input signature is a valid signature using the input public key for the hash of the current transaction
`OP_CHECKMULTISIG`	Checks that the *k* signatures on the transaction are valid signatures from *k* of the specified public keys.

Figure 3.6 a list of common Script instructions and their functionality.

Executing a script. To execute a script in a stack-based programming language, all we'll need is a stack that we can push data to and pop data from. We won't need any other memory or variables. That's what makes it so computationally simple. There are two types of instructions: data instructions and opcodes. When a data instruction appears in a script, that data is simply pushed onto the top of the stack. Opcodes, on the other hand, perform some function, often taking as input data that is on top of the stack.

Now let's look at how the Bitcoin script in Figure 3.5 is executed. Refer to Figure 3.7, where we show the state of the stack after each instruction. The first two instructions in this script are data instructions — the signature and the public key used to generate that signature. These were specified by the recipient in the scriptSig component, or the input script. As we mentioned, when we see a data instruction, we just push it onto the stack. The rest of the script is the part that was specified by the sender — the scriptPubKey component.

First we have the duplicate instruction, OP_DUP, so we just push a copy of the public key onto the top of the stack. The next instruction is OP_HASH160, which tells us to pop the top value, compute its cryptographic hash, and push the result onto the top of the stack. When this instruction finishes executing, we will have replaced the public key on the top of the stack with its hash.

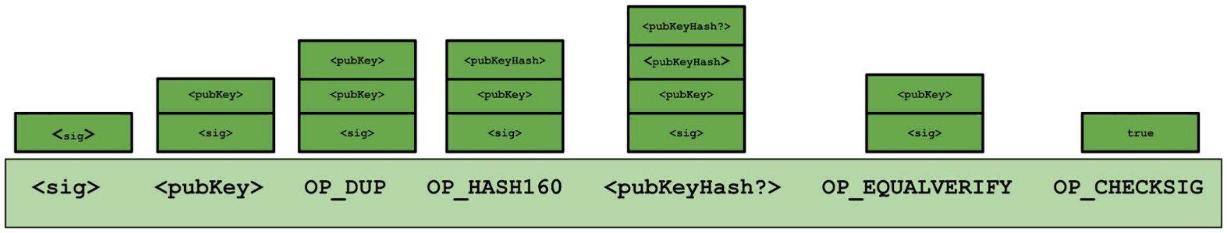


Figure 3.7 **Execution of a Bitcoin script.** On the bottom, we show the instruction in the script. Data instructions are denoted with surrounding angle brackets, whereas opcodes begin with "OP_". On the top, we show the stack just after that instruction has been executed.

Next, we're going to do one more push of data onto the stack. Recall that this data was specified by the sender of the coins. It is the hash of the public key that the sender specified had to be used to generate the signature to redeem these coins. At this point, there are two values at the top of the stack. There is the hash of the public key, as specified by the sender, and the hash of the public key that was used by the recipient when trying to claim the coins.

At this point we'll run the EQUALVERIFY command, which checks that the two values at the top of the stack are equal. If they aren't, an error will be thrown, and the script will stop executing. But in our example, we'll assume that they're equal, that is, that the recipient of the coins used the correct public key. That instruction will consume those two data items that are at the top of the stack, And the stack now contains two items — a signature and the public key that was used for that signature.

We've already checked that this public key was the public key that is required, and now we have to check if the signature is valid. This is a great example of where the Bitcoin scripting language is built with cryptography in mind. Even though it's a fairly simple language in terms of logic, there's some quite powerful instructions in there, like this "OP_CHECKSIG" instruction. This single instruction pops those two values off of the stack, and does the entire signature verification in one go.

But what is this actually a signature of? What was the input to the signature function? It turns out there's only one thing you can sign in Bitcoin — an entire transaction. So the "CHECKSIG" instruction pops the two values, the public key and signature, off the stack, and verifies that is a valid signature for the entire transaction using that public key. Now we've executed every instruction in the script, and there's nothing left on the stack. Provided there weren't any errors, the output of this script will simply be **true** indicating that the transaction is valid.

What's used in practice. In theory, Script lets us specify, in some sense, arbitrary conditions that must be met in order to spend coins. But, as of today, this flexibility isn't used very heavily. If we look at the scripts that have actually been used in the history of Bitcoin so far, the vast majority, 99.9 percent, are exactly the same script, which is in fact the script that we used in our example. As we saw, this script just specifies one public key and requires a signature for that public key in order to spend the coins.

There are a few other instructions that do get some use. MULTISIG gets used a little bit as does a special type of script called Pay-to-Script-Hash which we'll discuss shortly. But other than that, there hasn't been much diversity in terms of what scripts get used. This is because Bitcoin nodes, by default, have a whitelist of standard scripts, and they refuse to accept scripts that are not on the list. This doesn't mean that those scripts can't be used at all; it just makes them harder to use. In fact this distinction is a very subtle point which we'll return to in a bit when we talk about the Bitcoin peer-to-peer network.

Proof of burn. A proof-of-burn is a script that can never be redeemed. Sending coins to a proof-of-burn script establishes that they have been destroyed since there's no possible way for them to be spent. One use of proof-of-burn is to bootstrap an alternative to Bitcoin by forcing people to destroy Bitcoin in order to gain coins in the new system. We'll discuss this in more detail in Chapter 10. Proof-of-burn is quite simple to implement: the OP_RETURN opcode throws an error if it's ever reached. No matter what values you put before OP_RETURN, that instruction will get executed eventually, in which case this script will return false.

Because the error is thrown, the data in the script that comes after OP_RETURN will not be processed. So this is an opportunity for people to put arbitrary data in a script, and hence into the block chain. If, for some reason, you want to write your name, or if you want to timestamp and prove that you knew some data at a specific time, you can create a very low value Bitcoin transaction. You can destroy a very small amount of currency, but you get to write whatever you want into the block chain, which should be kept around forever.

Pay-to-script-hash. One consequence of the way that Bitcoin scripts works is that the sender of coins has to specify the script exactly. But this can sometimes be quite a strange way of doing things. Say, for example, you're a consumer shopping online, and you're about to order something. And you say, "Alright, I'm ready to pay. Tell me the address to which I should send my coins." Now, say that the company that you're ordering from is using MULTISIG addresses. Then, since the one spending the coins has to specify this, the retailer will have to come back and say, "Oh, well, we're doing something fancy now. We're using MULTISIG. We're going to ask you to send the coins to some complicated script." You might say, "I don't know how to do that. That's too complicated. As a consumer, I just want to send to a simple address."

In response to this problem, there's a really clever hack in Bitcoin. Instead of having the sender specify the entire script, the sender can specify just a hash of the script that is going to be needed to redeem those coins. So the sender just needs to specify a very simple script which just hashes the top value on the stack, and checks to see if it's equal to the required redemption script. The receiver of those coins needs to specify as a data value, the value of the script whose hash the sender specified. After this happens, a special second step of validation is going to occur. That top data value from the stack is going to be reinterpreted as instructions, and then it's going to be executed a second time as a script.

So we see there were two stages that happened here. First there was this traditional script which checked that the redemption script had the right hash. And then the redemption script will be de-serialized, and run as a script itself. And here's where the actual signature check is going to happen. This is called ***pay-to-script-hash*** in Bitcoin and it is often abbreviated as ***P2SH***. It is an alternative to the normal mode of operation, which is pay to a public key.

Getting support for P2SH was quite complicated since it wasn't part of Bitcoin's initial design specification. It was added after the fact. This is probably the most notable feature that's been added to Bitcoin that wasn't there in the original specification. And it solves a couple of important problems. It removes complexity from the sender, so the recipient can just specify a hash that the sender sends money to. In our example above, Alice need not worry that Bob is using multisig; she just sends to Bob's P2SH address, and it is Bob's responsibility to specify the fancy script when he wants to redeem the coins.

P2SH also has a nice efficiency gain. Miners have to track the set of output scripts that haven't been redeemed yet, and with P2SH outputs, the output scripts are now much smaller as they only specify a hash. All of the complexity is pushed to the input scripts.

3.3 Applications of Bitcoin scripts

Now that we understand how Bitcoin scripts work, let's take a look at some of the powerful applications that can be realized with this scripting language. It turns out we can do many neat things that will justify the complexity of having the scripting language instead of just specifying public keys.

Escrow transactions. Say Alice and Bob want to do business with each other — Alice wants to pay Bob in Bitcoin for Bob to send some physical goods to Alice. The problem though is that Alice doesn't want to pay until after she's received the goods, but Bob doesn't want to send the goods until after he has been paid. What can we do about that? A nice solution in Bitcoin that's been used in practice is to introduce a third party and do an escrow transaction.

Escrow transactions can be implemented quite simply using MULTISIG. Alice doesn't send the money directly to Bob, but instead creates a MULTISIG transaction that requires two of three people to sign in order to redeem the coins. And those three people are going to be Alice, Bob, and some third party arbitrator, Judy, who will come into play in case there's any dispute. So Alice creates a 2-of-3 MULTISIG transaction that sends some coins she owns and specifies that they can be spent if any two of Alice, Bob, and Judy sign. This transaction is included in the block chain, and at this point, these coins are held in escrow between Alice, Bob, and Judy, such that any two of them can specify where the coins should go. At this point, Bob is convinced that it's safe to send the goods over to Alice, so he'll mail them or deliver them physically. Now in the normal case, Alice and Bob are both honest. So, Bob will send over the goods that Alice is expecting, and when Alice receives the goods, Alice and Bob both sign a transaction redeeming the funds from escrow, and sending them to Bob. Notice that in this case where both Alice and Bob are honest, Judy never had to get involved at all. There was no dispute, and Alice's and Bob's signatures met the 2-of-3 requirement of the MULTISIG transaction. So in the normal case, this isn't that much less efficient than Alice just sending Bob the money. It requires just one extra transaction on the block chain.

But what would have happened if Bob didn't actually send the goods or they got lost in the mail? Or perhaps the goods were different than what Alice ordered? Alice now doesn't want to pay Bob because she thinks that she got cheated, and she wants to get her money back. So Alice is definitely not going to sign a transaction that releases the money to Bob. But Bob also may deny any wrongdoing and refuse to sign a transaction that releases the money back to Alice. This is where Judy needs to get involved. Judy's going to have to decide which of these two people deserves the money. If Judy decides that Bob cheated, Judy will be willing to sign a transaction along with Alice, sending the money from escrow back to Alice. Alice's and Judy's signatures meet the 2-of-3 requirement of the MULTISIG transaction, and Alice will get her money back. And, of course, if Judy thinks that Alice is at fault here, and Alice is simply refusing to pay when she should, Judy can sign a transaction along with Bob, sending the money to Bob. So Judy decides between the two possible outcomes. But the nice thing is that she won't have to be involved unless there's a dispute.

Green addresses. Another cool application is what are called green addresses. Say Alice wants to pay Bob, and Bob's offline. Since he's offline, Bob can't go and look at the block chain to see if a transaction that Alice is sending is actually there. It's also possible that Bob is online, but doesn't have the time to go and look at the block chain and wait for the transactions to be confirmed. Remember that normally we want a transaction to be in the block chain and be confirmed by six blocks, which takes up to an hour, before we trust that it's really in the block chain. But for some merchandise such as food, Bob can't wait an hour before delivering. If Bob were a street vendor selling hot dogs, it's unlikely that Alice would wait around for an hour to receive her food. Or maybe Bob for some other reason doesn't have any connection to the internet at all, and is thus not going to be able to check the block chain.

To solve this problem of being able to send money using Bitcoin without the recipient being able to access the block chain, we have to introduce another third party, which we'll call the bank (in practice it could be an exchange or any other financial intermediary). Alice is going to talk to her bank, and say, "Hey, it's me, Alice. I'm your loyal customer. Here's my card or my identification. And I'd really like to pay Bob here, could you help me out?" And the bank will say, "Sure. I'm going to deduct some money out of your account. And draw up a transaction from one of my green addresses over to Bob."

So notice that this money is coming directly from the bank to Bob. Some of the money, of course, might be in a change address going back to the bank. But essentially, the bank is paying Bob here from a bank-controlled address, which we call a green address. Moreover, the bank guarantees that it will not double-spend this money. So as soon as Bob sees that this transaction is signed by the bank, if he trusts the bank's guarantee not to double-spend the money, he can accept that that money will eventually be his when it's confirmed in the block chain.

Notice that this is not a Bitcoin-enforced guarantee. This is a real-world guarantee, and in order for this system to work, Bob has to trust that the bank, in the real world, cares about their reputation, and won't double-spend for that reason. And the bank will be able to say, "You can look at my history. I've been using this green address for a long time, and I've never double spent. Therefore I'm very unlikely to do so in the future." Thus Bob no longer has to trust Alice, whom he may know nothing about. Instead, he places his trust in the bank that they will not double-spend the money that they sent him.

Of course, if the bank ever does double-spend, people will stop trusting its green address(es). In fact, the two most prominent online services that implemented green addresses were Instawallet and Mt. Gox, and both ended up collapsing. Today green addresses aren't used very much. When the idea was first proposed, it generated much excitement as a way to do payments more quickly and without accessing the block chain. Now, however, people have become quite nervous about the idea and are worried that it puts too much trust in the bank.

Efficient micro-payments. A third example of Bitcoin scripts is a way to do efficient micro-payments. Say that Alice is a customer who wants to continually pay Bob small amounts of money for some service that Bob provides. For example, Bob may be Alice's wireless service provider, and requires her to pay a small fee for every minute that she talks on her phone.

Creating a Bitcoin transaction for every minute that Alice speaks on the phone won't work. That will create too many transactions, and the transaction fees add up. If the value of each one of these transactions is on the order of what the transaction fees are, Alice is going to be paying quite a high cost to do this.

What we'd like is to be able to combine all these small payments into one big payment at the end. It turns out that there's a neat way to do this. We start with a MULTISIG transaction that pays the maximum amount Alice would ever need to spend to an output requiring both Alice and Bob to sign to release the coins. Now, after the first minute that Alice has used the service, or the first time Alice needs to make a micropayment, she signs a transaction spending those coins that were sent to the MULTISIG address, sending one unit of payment to Bob and returning the rest to Alice. After the next minute of using the service, Alice signs another transaction, this time paying two units to Bob and sending the rest to herself. Notice these are signed only by Alice, and haven't been signed by Bob yet, nor are they being published to the block chain. Alice will keep sending these transactions to Bob every minute that she uses the service. Eventually, Alice will finish using the service, and tells Bob, "I'm done, please cut off my service." At this point Alice will stop signing additional transactions. Upon hearing this, Bob will say "Great. I'll disconnect your service, and I'll take that last transaction that you sent me, sign it, and publish that to the block chain."

Since each transaction was paying Bob a little bit more, and Alice a little bit less, the final transaction that Bob redeems pays him in full for the service that he provided and returns the rest of the money to Alice. All those transactions that Alice signed along the way won't make it to the block chain. Bob doesn't have to sign them. They'll just get discarded.

Technically all of these transactions are double-spends. So unlike the case with green addresses where we were specifically trying to avoid double-spends, with a strong guarantee, with this micro-payment protocol, we're actually generating a huge amount of potential double-spends. In practice, however, if both parties are operating normally, Bob will never sign any transaction but the last one, in which case the block chain won't actually see any attempt at a double-spend.

There's one other tricky detail: what if Bob never signs the last transaction? He may just say, "I'm happy to let the coins sit there in escrow forever," in which case, maybe the coins won't move, but Alice will lose the full value that she paid at the beginning. There's a very clever way to avoid this problem using a feature that we mentioned briefly earlier, and will explain now.

Lock time. To avoid this problem, before the micro-payment protocol can even start, Alice and Bob will both sign a transaction which refunds all of Alice's money back to her, but the refund is "locked" until some time in the future. So after Alice signs, but before she broadcasts, the first MULTISIG transaction that puts her funds into escrow, she'll want to get this refund transaction from Bob and hold on to it. That guarantees that if she makes it to time *t* and Bob hasn't signed any of the small transactions that Alice has sent, Alice can publish this transaction which refunds all of the money directly to her.

What does it mean that it's locked until time *t*? Recall when we looked at the metadata in Bitcoin transactions, that there was this lock_time parameter, which we had left unexplained. The way it works is that if you specify any value other than zero for the lock time, it tells miners not to publish the transaction until the specified lock time. The transaction will be invalid before either a specific block number, or a specific point in time, based on the timestamps that are put into blocks. So this is a way of preparing a transaction that can only be spent in the future if it isn't already spent by then. It works quite nicely in the micro-payment protocol as a safety valve for Alice to know that if Bob never signs, eventually she'll be able to get her money back.

Hopefully, these examples have shown you that we can do some neat stuff with Bitcoin scripts. We discussed three simple and practical examples, but there are many others that have been researched. One of them is multi-player lotteries, a very complicated multi-step protocol with lots of transactions having different lock times and escrows in case people cheat. There are also some neat protocols that utilize the scripting language to allow different people to get their coins together and mix them, so that it's harder to trace who owns which coin. We'll see that in detail in Chapter 6.

Smart contracts. The general term for contracts like the ones we saw in this section is smart contracts. These are contracts for which we have some degree of technical enforcement in Bitcoin, whereas traditionally they are enforced through laws or courts of arbitration. It's a really cool feature of Bitcoin that we can use scripts, miners, and transaction validation to realize the escrow protocol or the micro-payment protocol without needing a centralized authority.

Research into smart contracts goes far beyond the applications that we saw in this section. There are many types of smart contracts which people would like to be able to enforce but which aren't supported by the Bitcoin scripting language today. Or at least, nobody has come up with a creative way to implement them. As we saw, with a bit of creativity you can do quite a lot with the Bitcoin script as it currently stands.

3.4 Bitcoin blocks

So far in this chapter we've looked at how individual transactions are constructed and redeemed. But as we saw in chapter 2, transactions are grouped together into blocks. Why is this? Basically, it's an optimization. If miners had to come to consensus on each transaction individually, the rate at which new transactions could be accepted by the system would be much lower. Also, a hash chain of blocks is much shorter than a hash chain of transactions would be, since a large number of transactions can be put into each block. This will make it much more efficient to verify the block chain data structure.

The block chain is a clever combination of two different hash-based data structures. The first is a hash chain of blocks. Each block has a block header, a hash pointer to some transaction data, and a hash pointer to the previous block in the sequence. The second data structure is a per-block tree of all of the transactions that are included in that block. This is a Merkle tree and allows us to have a digest of all the transactions in the block in an efficient way. As we saw in Chapter 1, to prove that a transaction is included in a specific block, we can provide a path through the tree whose length is logarithmic in the number of transactions in the block. To recap, a block consists of header data followed by a list of transactions arranged in a tree structure.

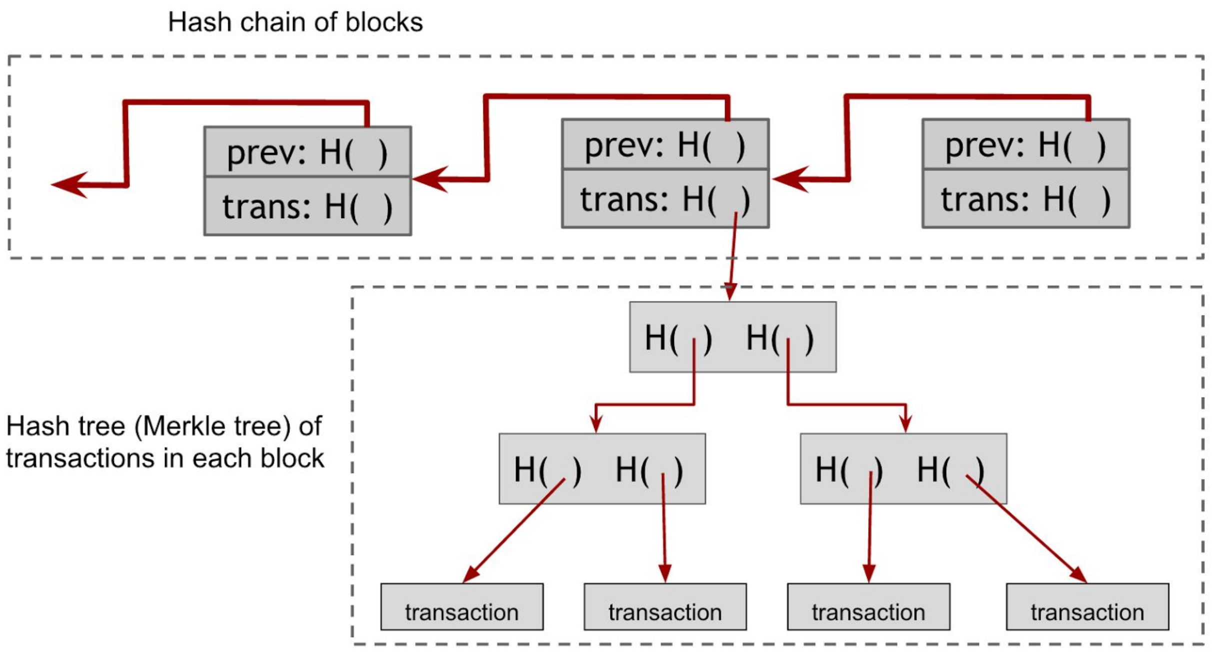


Figure 3.8. The Bitcoin block chain contains two different hash structures. The first is a hash chain of blocks that links the different blocks to one another. The second is internal to each block and is a Merkle Tree of transactions within the blocks.

The header mostly contains information related to the mining puzzle which we briefly discussed in the previous chapter and will revisit in Chapter 5. Recall that the hash of the block header has to start with a large number of zeros for the block to be valid. The header also contains a “nonce” that miners can change, a time stamp, and “bits”, which is an indication of how difficult this block was to find. The header is the only thing that’s hashed during mining. So to verify a chain of blocks, all we need to do is look at the headers. The only transaction data that’s included in the header is the root of the transaction tree — the “mrkl_root” field.

```
"in":[
      {
        "prev_out":{
          "hash":"000000.....0000000",
          "n":4294967295
        },
         "coinbase":"..."
      },
     "out":[
  {
    "value":"25.03371419",
    "scriptPubKey":"OPDUP OPHASH160 ... ”
         }
```

Figure 3.9. **coinbase transaction** A coinbase transaction creates new coins. It does not redeem a previous output, and it has a null hash pointer indicating this. It has a coinbase parameter which can contain arbitrary data. The value of the coinbase transaction is the block reward plus all of the transaction fees included in this block.

Another interesting thing about blocks is that they have a special transaction in the Merkle tree called the “coinbase” transaction. This is analogous to CreateCoins in Scroogecoin. So this is where the creation of new coins in Bitcoin happens. It mostly looks like a normal transaction but with several differences: (1) it always has a single input and a single output, (2) the input doesn’t redeem a previous output and thus contains a null hash pointer, since it is minting new bitcoins and not spending existing coins, (3) the value of the output is currently a little over 25 Bitcoins. The output value is the miner’s revenue from the block. It consists of two components: a flat mining reward, which is set by the system and which halves every 210,000 blocks (about 4 years), and the transaction fees collected from every transaction included in the block. (4) There is a special “coinbase” parameter, which is completely arbitrary — miners can put whatever they want in there.

Famously, in the very first block ever mined in Bitcoin, the coinbase parameter referenced a story in the Times of London newspaper involving the Chancellor bailing out banks. This has been interpreted as political commentary on the motivation for starting Bitcoin. It also serves as a sort of proof that the first block was mined after the story came out on January 3, 2009. One way in which the coinbase parameter has since been used is to signal support by miners for different new features.

To get a better feel for the block format and transaction format, the best way is to explore the block chain yourself. There are many websites that make this data accessible, such as blockchain.info. You can look at the graph of transactions, see which transactions redeem which other transactions, look for transactions with complicated scripts, and look at the block structure and see how blocks refer to other blocks. Since the block chain is a public data structure, developers have built pretty wrappers to explore it graphically.

3.5 The Bitcoin network

So far we've been talking about the ability for participants to publish a transaction and get it into the block chain as if this happens by magic. In fact this happens through the Bitcoin network. It's a peer-to-peer network, and it inherits many ideas from peer-to-peer networks that have been proposed for all sorts of other purposes. In the Bitcoin network, all nodes are equal. There is no hierarchy, and there are no special nodes or master nodes. It runs over TCP and has a random topology, where each node peers with other random nodes. New nodes can join at any time. In fact, you can download a Bitcoin client today, spin up your computer as a node, and it will have equal rights and capabilities as every other node on the Bitcoin network.

The network changes over time and is quite dynamic due to nodes entering and leaving. There isn't an explicit way to leave the network. Instead, if a node hasn't been heard from in a while — three hours is the duration that's hardcoded into the common clients — other nodes start to forget it. In this way, the network gracefully handles nodes going offline.

Recall that nodes connect to random peers and there is no geographic topology of any sort. Now say you launch a new node and want to join the network. You start with a simple message to one node that you know about. This is usually called your ***seed node***, and there are a few different ways you can look up lists of seed nodes to try connecting to. You send a special message, saying, "Tell me the addresses of all the other nodes in the network that you know about." You can repeat the process with the new nodes you learn about as many times as you want. Then you can choose which ones to peer with, and you'll be a fully functioning member of the Bitcoin network. There are several steps that involve randomness, and the ideal outcome is that you're peered with a random set of nodes. To join the network, all you need to know is how to contact one node that's already on the network.

What is the network good for? To maintain the block chain, of course. So to publish a transaction, we want to get the entire network to hear about it. This happens through a simple ***flooding*** algorithm, sometimes called a ***gossip protocol***. If Alice wants to pay Bob some money, her client creates and her node sends this transaction to all the nodes it's peered with. Each of those nodes executes a series of checks to determine whether or not to accept and relay the transaction. If the checks pass, the node in turn sends it to all of its peer nodes. Nodes that hear about a transaction put it in a pool of transactions which they've heard about but that aren't on the block chain yet. If a node hears about a transaction that's already in its pool, it doesn't further broadcast it. This ensures that the flooding protocol terminates and transactions don't loop around the network forever. Remember that every transaction is identified uniquely by its hash, so it's easy to look up a transaction in the pool.

When nodes hear about a new transaction, how do they decide whether or not they should propagate it? There are four checks. The first and most important check is transaction validation — the transaction must be valid with the current block chain. Nodes run the script for each previous output being redeemed and ensure that the scripts return true. Second, they check that the outputs being redeemed here haven't already been spent. Third, they won't relay an already-seen transaction, as mentioned earlier. Fourth, by default, nodes will only accept and relay "standard" scripts based on a small whitelist of scripts.

All these checks are just sanity checks. Well-behaving nodes all implement these to try to keep the network healthy and running properly, but there's no rule that says that nodes have to follow these specific steps. Since it's a peer-to-peer network, and anybody can join, there's always the possibility that a node might forward double-spends, non-standard transactions, or outright invalid transactions. That's why every node must do the checking for itself.

Since there is latency in the network, it's possible that nodes will end up with a different view of the pending transaction pool. This becomes particularly interesting and important when there is an attempted double-spend. Let's say Alice attempts to pay the same bitcoin to both Bob and Charlie, and sends out two transactions at roughly the same time. Some nodes will hear about the Alice → Bob transaction first while others will hear about the Alice → Charlie transaction first. When a node hears either of these transactions, it will add it to its transaction pool, and if it hears about the other one later it will look like a double-spend. The node will drop the latter transaction and won't relay it or add it to its transaction pool. As a result, the nodes will temporarily disagree on which transactions should be put into the next block. This is called a race condition.

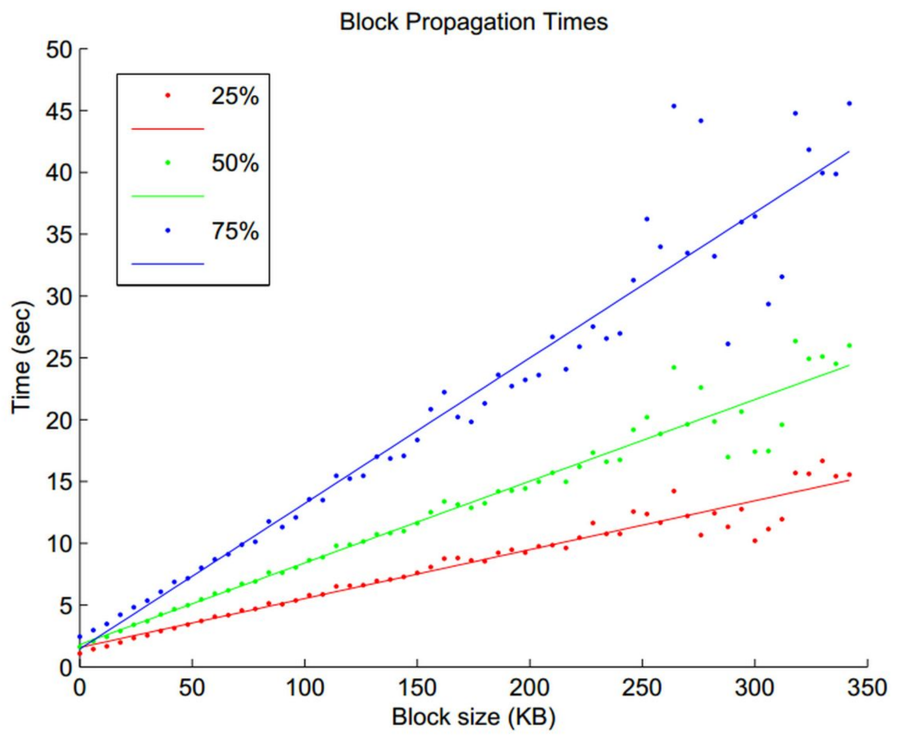
The good news is that this is perfectly okay. Whoever mines the next block will essentially break the tie and decide which of those two pending transactions should end up being put permanently into a block. Let's say the Alice → Charlie transaction makes it into the block. When nodes with the Alice → Bob transaction hear about this block, they'll drop the transaction from their memory pools because it is a double-spend. When nodes with the Alice → Charlie transaction hear about this block, they'll drop the transaction from their memory pools because it's already made it into the block chain. So there will be no more disagreement once this block propagates to the network.

Since the default behavior is for nodes to hang onto whatever they hear first, network position matters. If two conflicting transactions or blocks get announced at two different positions in the network, they'll both begin to flood throughout the network and which transaction a node sees first will depend on where it is in the network.

Of course this assumes that every node implements this logic where they keep whatever they hear first. But there's no central authority enforcing this, and nodes are free to implement any other logic they want for choosing which transactions to forward. We'll take up this question in Chapter 5 and discuss why miners, in particular, might want to implement some different logic other than the default behavior.

So far we've been mostly discussing propagation of transactions. The logic for announcing new blocks, whenever miners find a new block, is almost exactly the same as propagating a new transaction and it is all subject to the same race conditions. If two valid blocks are mined at the same time, only one of these can be included in the long term consensus chain. Ultimately, which of these blocks will be included will depend on which blocks the other nodes build on top of, and the one that does not get into the consensus chain will be orphaned.

Validating a block is more complex than validating transactions. In addition to validating the header and making sure that the hash value is in the acceptable range, nodes must validate every transaction included in the block. Finally, a node will forward a block only if it builds on the longest branch, based on its perspective of what the block chain (which is really a tree of blocks) looks like. This avoids forks building up. But just like with transactions, nodes can implement different logic if they want — they may relay blocks that aren't valid or blocks that build off of an earlier point in the block chain. This would build a fork, but that's okay. The protocol is designed to withstand that.



Source: Yonatan Sompolinsky and Aviv Zohar: "Accelerating Bitcoin's Transaction Processing" 2014

Figure 3.10 **Block propagation time.** This graph shows the average time that it takes a block to reach various percentages of the nodes in the network.

What is the latency of the flooding algorithm? The graph in Figure 3.10 shows the average time for new blocks to propagate to every node in the network. The three lines show the 25th, the 50th, and the 75th percentile block propagation time. As you can see, propagation time is basically proportional to the size of the block. This is because network bandwidth is the bottleneck. The larger blocks take over 30 seconds to propagate to most nodes in the network. So it isn't a particularly efficient protocol. On the Internet, 30 seconds is a pretty long time. In Bitcoin's design, having a simple network with little structure where nodes are equal and can come and go at any time took priority over efficiency. So a block may need to go through many nodes before it reaches the most distant nodes in the network. If the network were instead designed top-down for efficiency, we could make sure that the path between any two nodes is short.

Size of the network. It is difficult to measure how big the network is since it is dynamic and there is no central authority. A number of researchers have come up with estimates. On the high end, some say that over a million IP addresses in a given month will, at some point, act, at least temporarily, as a Bitcoin node. On the other hand, there seem to be only about 5,000 to 10,000 nodes that are permanently connected and fully validate every transaction they hear. This may seem like a surprisingly low number, but as of this writing there is no evidence that the number of fully validating nodes is going up, and it may in fact be dropping.

Storage requirements. Fully validating nodes must stay permanently connected so as to hear about all the data. The longer a node is offline, the more catching up it will have to do when it rejoins the network. Such nodes also have to store the entire block chain and need a good network connection to be able to hear every new transaction and forward it to peers. The storage requirement is currently in the low tens of gigabytes (see Figure 3.11), well within the abilities of a single commodity desktop machine.

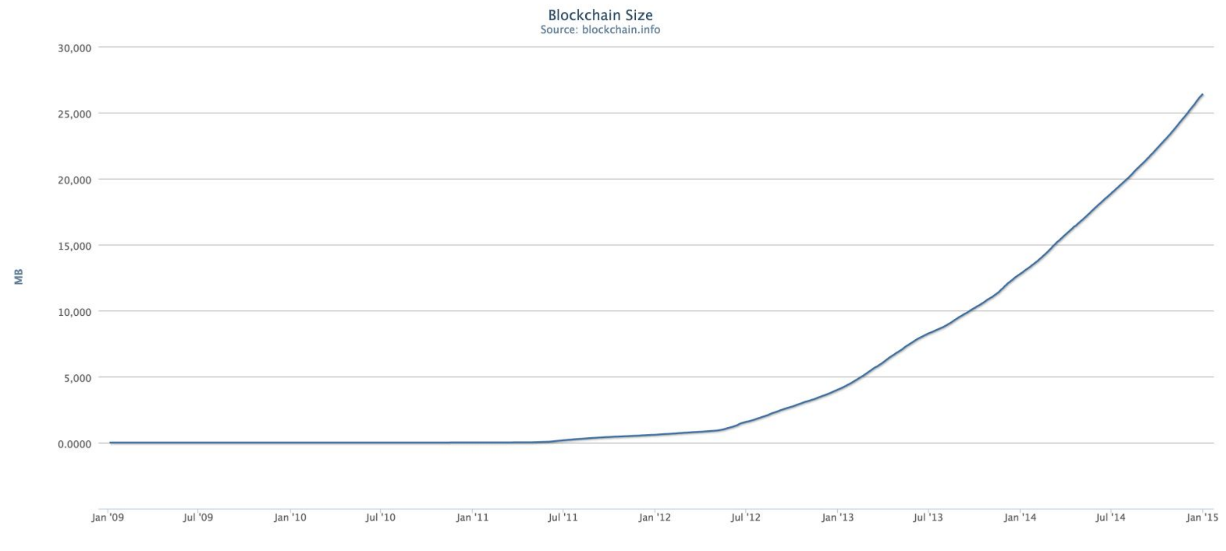


Figure 3.11. **Size of the block chain.** Fully validating nodes must store the entire block chain, which as of the end of 2014 is over 26 gigabytes.

Finally, fully validating nodes must maintain the entire set of unspent transaction outputs, which are the coins available to be spent. Ideally this should be stored in RAM, so that upon hearing a new proposed transaction on the network, the node can quickly look up the transaction outputs that it's attempting to claim, run the scripts, see if the signatures are valid, and add the transaction to the transaction pool. As of mid-2014, there are over 44 million transactions on the block chain of which 12 million are unspent. Fortunately, that's still small enough to fit in less than a gigabyte of RAM in an efficient data structure.

Lightweight nodes. In contrast to fully validating nodes, there are lightweight nodes, also called thin clients or Simple Payment Verification (SPV) clients. In fact, the vast majority of nodes on the Bitcoin network are lightweight nodes. These differ from fully validating nodes in that they don't store the entire block chain. They only store the pieces that they need to verify specific transactions that they care about. If you use a wallet program, it would typically incorporate an SPV node. The node downloads the block headers and transactions that represent payments to your addresses.

An SPV node doesn't have the security level of a fully validating node. Since the node has block headers, it can check that the blocks were difficult to mine, but it can't check to see that every transaction included in a block is actually valid because it doesn't have the transaction history and doesn't know the set of unspent transactions outputs. SPV nodes can only validate the transactions that actually affect them. So they're essentially trusting the fully validating nodes to have validated all the other transactions that are out there. This isn't a bad security trade off. They're assuming there are fully validating nodes out there that are doing the hard work, and that if miners went through the trouble to mine this block, which is a really expensive process, they probably also did some validation to make sure that this block wouldn't be rejected.

The cost savings of being an SPV node are huge. The block headers are only about 1/1,000 the size of the block chain. So instead of storing a few tens of gigabytes, it's only a few tens of megabytes. Even a smartphone can easily act as an SPV node in the Bitcoin network.

Since Bitcoin rests on an open protocol, ideally there would be many different implementations that interact with each other seamlessly. That way if there's a bad bug in one, it's not likely to bring down the entire network. The good news is that the protocol has been successfully re-implemented. There are implementations in C++ and Go, and people are working on quite a few others. The bad news is that most of the nodes on the network are running the bitcoind library, written in C++, maintained by the Bitcoin core developers, and some of these nodes are running previous out-of-date versions that haven't been updated. In any event, most are running some variation of this one common client.

3.6 Limitations and improvements

Finally, we'll talk about some built-in limitations to the Bitcoin protocol, and why it's challenging to improve them. There are many constraints hard-coded into the Bitcoin protocol, which were chosen when Bitcoin was proposed in 2009, before anyone really had any idea that it might grow into a globally-important currency. Among them are the limits on the average time per block, the size of blocks, the number of signature operations in a block, and the divisibility of the currency, the total number of Bitcoins, and the block reward structure.

The limitations on the total number of Bitcoins in existence, as well as the structure of the mining rewards are very likely to never be changed because the economic implications of changing them are too great. Miners and investors have made big bets on the system assuming that the Bitcoin reward structure and the limited supply of Bitcoins will remain the way it was planned. If that changes, it will have large financial implications for people. So the community has basically agreed that those aspects, whether or not they were wisely chosen, will not change.

There are other changes that would seem to make everybody better off, because some initial design choices don't seem quite right with the benefit of hindsight. Chief among these are limits that affect the throughput of the system. How many transactions can the Bitcoin network process per second? This limitation comes from the hard coded limit on the size of blocks. Each block is limited to a megabyte, about a million bytes. Each transaction is at least 250 bytes. Dividing 1,000,000 by 250, we see that each block has a limit of 4,000 transactions, and given that blocks are found about every 10 minutes, we're left with about 7 transactions per second, which is all that the Bitcoin network can handle. It may seem that changing these limits would be a matter of tweaking a constant in a source code file somewhere. However, it's really hard to effect such a change in practice, for reasons that we will explain shortly.

So how does seven transactions per second compare? It's quite low compared to the throughput of any major credit card processor. Visa's network is said to handle about 2,000 transactions per second around the world on average, and capable of handling 10,000 transactions per second during busy periods. Even Paypal, which is newer and smaller than Visa, can handle 100 transactions per second at peak times. That's an order of magnitude more than Bitcoin.

Another limitation that people are worried about in the long term is that the choices of cryptographic algorithms in Bitcoin are fixed. There are only a couple of hash algorithms available, and only one signature algorithm, ECDSA, over a specific elliptic curve called secp256k1. There's some concern that over the lifetime of Bitcoin — which people hope will be very long — this algorithm might be broken. Cryptographers might come up with a clever new attack that we haven't foreseen which makes the algorithm insecure. The same is true of the hash functions; in fact, in the last decade hash functions have seen steady progress in cryptanalysis. SHA-1, which is included in Bitcoin, already has some known cryptographic weaknesses, albeit not fatal. To change this, we would have to extend the Bitcoin scripting language to support new cryptographic algorithms.

Changing the protocol. How can we go about introducing new features into the Bitcoin protocol? You might think that this is simple — just release a new version of the software, and tell all nodes to upgrade. In reality, though, this is quite complicated. In practice, it's impossible to assume that every node would upgrade. Some nodes in the network would fail to get the new software or fail to get it in time. The implications of having most nodes upgrade while some nodes are running the old version depends very much on the nature of the changes in the software. We can differentiate between two types of changes: those that would cause a ***hard fork*** and those that would cause a ***soft fork***.

Hard forks. One type of change that we can make introduces new features that were previously considered invalid. That is, the new version of the software would recognize blocks as valid that the old software would reject. Now consider what happens when most nodes have upgraded, but some have not. Soon the longest branch will contain blocks that are considered invalid by the old nodes. So the old nodes will go off and work on a branch of the block chain that excludes blocks with the new feature. Until they upgrade their software, they'll consider their (shorter) branch to be the longest valid branch.

This type of change is called a hard forking change because it makes the block chain split. Every node in the network will be on one or the other side of it based on which version of the protocol it's running. Of course, the branches will never join together again. This is considered unacceptable by the community since old nodes would effectively be cut out of the Bitcoin network if they don't upgrade their software.

Soft forks. A second type of change that we can make to Bitcoin is adding features that make validation rules stricter. That is, they restrict the set of valid transactions or the set of valid blocks such that the old version would accept all of the blocks, whereas the new version would reject some. This type of change is called a soft fork, and it can avoid the permanent split that a hard fork introduces.

Consider what happens when we introduce a new version of the software with a soft forking change. The nodes running the new software will be enforcing some new, tighter, set of rules. Provided that the majority of nodes switch over to the new software, these nodes will be able to enforce the new rules. Introducing a soft fork relies on enough nodes switching to the new version of the protocol that they'll be able to enforce the new rules, knowing that the old nodes won't be able to enforce the new rules because they haven't heard of them yet.

There is a risk that old miners might mine invalid blocks because they include some transactions that are invalid under the new, stricter, rules. But the old nodes will at least figure out that some of their blocks are being rejected, even if they don't understand the reason. This might prompt their operators to upgrade their software. Furthermore, if their branch gets overtaken by the new miners, the old miners switch to it. That's because blocks considered valid by new miners are also considered valid by old miners. Thus, there won't be a hard fork; instead, there will be many small, temporary forks.

The classic example of a change that was made via soft fork is pay-to-script-hash, which we discussed earlier in this chapter. Pay-to-script-hash was not present in the first version of the Bitcoin protocol. This is a soft fork because from the view of the old nodes, a valid pay-to-script-hash transaction would still verify correctly. As interpreted by the old nodes, the script is simple — it hashes one data value and checks if the hash matches the value specified in the output script. Old nodes don't know to do the (now required) additional step of running that value itself to see if it is a valid script. We rely on new nodes to enforce the new rules, i.e. that the script actually redeems this transaction.

So what could we possibly add with a soft fork? Pay-to-script-hash was successful. It's also possible that new cryptographic schemes could be added by a soft fork. We could also add some extra metadata in the coinbase parameter that had some meaning. Today, any value is accepted in the coinbase parameter. But we could, in the future, say that the coinbase has to have some specific format. One idea that's been proposed is that, in each new block, the coinbase includes the Merkle root of a tree containing the entire set of unspent transactions. It would only result in a soft fork, because old nodes might mine a block that didn't have the required new coinbase parameter that got rejected by the network, but they would catch up and join the main chain that the network is mining.

Other changes might require a hard fork. Examples of this are adding new opcodes to Bitcoin, changing the limits on block or transactions size, or various bug fixes. Fixing the bug we discussed earlier, where the MULTISIG instruction pops an extra value off the stack, would actually require a hard fork. That explains why, even though it's an annoying bug, it's much easier to leave it in the protocol and have people work around it rather than have a hard fork change to Bitcoin. Hard forking changes, even though they would be nice, are very unlikely to happen within the current climate of Bitcoin. But many of these ideas have been tested out and proved to be successful in alternative cryptocurrencies, which start over from scratch. We'll be talking about those in a lot more detail in Chapter 10.

Online resources. In this chapter, we discussed a lot of technical details, and you may find it difficult to absorb them all at once. To supplement the material in this chapter, it's useful to go online and see some of the things we discussed in practice. There are numerous websites that allow you to examine blocks and transactions and see what they look like. One such "blockchain explorer" is the website blockchain.info.

At this point, you should be familiar with the technical mechanics of Bitcoin and how a Bitcoin node operates. But, human beings aren't Bitcoin nodes, and you're never going to run a Bitcoin node in your head. So how do you, as a human, actually interact with this network to get it to be useable as a currency? How do you find a node to inform about your transaction? How do you get Bitcoins in exchange for cash? How do you store your Bitcoins? All of these questions are crucial for building a currency that will actually work for people, as opposed to just software, and we will answer these questions in the next chapter.

Exercises

1. **Transaction validation**: Consider the steps involved in processing Bitcoin transactions. Which of these steps are computationally expensive? If you're an entity validating many transactions (say, a miner) what data structure might you build to help speed up verification?
2. **Bitcoin script**: For the following questions, you're free to use non-standard transactions and op codes that are currently disabled. You can use <data> as a shorthand to represent data values pushed onto the stack. For a quick reference, see here: https://en.bitcoin.it/wiki/Script.
    a. Write the Bitcoin ScriptPubKey script for a transaction that can be redeemed by anybody who supplies a square root of 1764.
    b. Write a corresponding ScriptSig script to redeem your transaction.
    c. Suppose you wanted to issue a new RSA factoring challenge by publishing a transaction that can be redeemed by anybody who can factor a 1024-bit RSA number (RSA numbers are the product of two large, secret prime numbers). What difficulties might you run into?
3. **Bitcoin script II**: Alice is backpacking and is worried about her devices containing private keys getting stolen. So she would like to store her bitcoins in such a way that they can be redeemed via knowledge of only a password. Accordingly, she stores them in the following ScriptPubKey address:
    OP_SHA1
    <0x084a3501edef6845f2f1e4198ec3a2b81cf5c6bc>
    OP_EQUALVERIFY
    a. Write a ScriptSig script that will successfully redeem this transaction. [Hint: it should only be one line long.]
    b. Explain why this is not a secure way to protect Bitcoins using a password.
    c. Would implementing this using Pay-to-script-hash (P2SH) fix the security issue(s) you identified? Why or why not?
4. **Bitcoin script III.**
    a. Write a ScriptPubKey that requires demonstrating a SHA-256 collision to redeem.
    b. (Hard) write a corresponding ScriptSig that will successfully redeem this transaction.
5. **Burning and encoding**
    a. What are some ways to burn bitcoins, i.e., to make a transaction unredeemable? Which of these allow a proof of burn, i.e., convincing any observer that no one can redeem such a transaction?
    b. What are some ways to encode arbitrary data into the block chain? Which of these result in burnt bitcoins?
    [Hint: you have more control over the contents of the transaction "out" field than might at first appear.]

   c. One user encoded some JavaScript code into the block chain. What might have been a motivation for doing this?

6. **Green addresses:** One problem with green addresses is that there is no punishment against double-spending within the Bitcoin system itself. To solve this, you decide to design an altcoin called "GreenCoin" that has built-in support for green addresses. Any attempt at double spending from addresses (or transaction outputs) that have been designated as "green" must incur a financial penalty in a way that can be enforced by miners. Propose a possible design for GreenCoin.
7. **SPV proofs**: Suppose Bob the merchant runs a lightweight client and receives the current head of the block chain from a trusted source.
   a. What information should Bob's customers provide to prove that their payment to Bob has been included in the block chain? Assume Bob requires 6 confirmations.
   b. Estimate how many bytes this proof will require. Assume there are 1024 transactions in each block.
8. **Adding new features**: Assess whether the following new features could be added using a hard fork or a soft fork:
   a. Adding a new OP_SHA3 script instruction
   b. Disabling the OP_SHA1 instruction
   c. A requirement that each miner include a Merkle root of unspent transaction outputs (UTXOs) in each block
   d. A requirement that all transactions have their outputs sorted by value in ascending order
9. **More forking**
   a. The most prominent Bitcoin hard fork was a transient one caused by the version 0.8 bug. How many blocks were abandoned when the fork was resolved?
   b. The most prominent Bitcoin soft fork was the addition of pay-to-script-hash. How many blocks were orphaned because of it?
   c. Bitcoin clients go into "safe mode" when they detect that the chain has forked. What heuristic(s) could you use to detect this?

Bitcoin and Cryptocurrency Technologies

Arvind Narayanan, Joseph Bonneau, Edward Felten,
Andrew Miller, Steven Goldfeder

Draft — Oct 6, 2015

Feedback welcome! Email bitcoinbook@lists.cs.princeton.edu

Chapter 4: How to Store and Use Bitcoins

This chapter is about how we store and use bitcoins in practice.

4.1 Simple Local Storage

Let's begin with the simplest way of storing bitcoins, and that is simply putting them on a local device. As a recap, to spend a bitcoin you need to know some public information and some secret information. The public information is what goes on the block chain — the identity of the coin, how much it's worth, and so on. The secret information is the secret key of the owner of the bitcoin, presumably, that's you. You don't need to worry too much about how to store the public information because you can always get it back when you need to. But the secret signing key is something you'd better keep track of. So in practice storing your bitcoins is all about storing and managing your keys.

Storing bitcoins is really all about storing and managing Bitcoin secret keys.

When figuring out how to store and manage keys, there are three goals to keep in mind. The first is availability: being able to actually spend your coins when you want to. The second is security: making sure that nobody else can spend your coins. If someone gets the power to spend your coins they could just send your coins to themselves, and then you don't have the coins anymore. The third goal is convenience, that is, key management should be relatively easy to do. As you can imagine, achieving all three simultaneously can be a challenge.

Different approaches to key management offer different trade-offs between availability, security, and convenience.

The simplest key management method is storing them on a file on your own local device: your computer, your phone, or some other kind of gadget that you carry, or own, or control. This is great for convenience: having a smartphone app that allows spending coins with the push of a few buttons is hard to beat. But this isn't great for availability or security — if you lose the device, if the device crashes, and you have to wipe the disc, or if your file gets corrupted, your keys are lost, and so are your coins. Similarly for security: if someone steals or breaks into your device, or it gets infected with malware, they can copy your keys and then they can then send all your coins to themselves.

In other words, storing your private keys on a local device, especially a mobile device, is a lot like carrying around money in your wallet or in your purse. It's useful to have some spending money, but you don't want to carry around your life savings because you might lose it, or somebody might steal it. So what you typically do is store a little bit of information/a little bit of money in your wallet, and keep most of your money somewhere else.

Wallets. If you're storing our bitcoins locally, you'd typically use wallet software, which is software that keeps track of all your coins, manages all the details of your keys, and makes things convenient with a nice user interface. If you want to send $4.25 worth of bitcoins to your local coffee shop the wallet software would give you some easy way to do that. Wallet software is especially useful because you typically want to use a whole bunch of different addresses with different keys associated with them. As you may remember, creating a new public/private key pair is easy, and you can utilize this to improve your anonymity or privacy. Wallet software gives you a simple interface that tells you how much is in your wallet. When you want to spend bitcoins, it handles the details of which keys to use and how to generate new addresses and so on.

Encoding keys: base 58 and QR codes. To spend or receive bitcoins, you also need a way to exchange an address with the other party — the address to which bitcoins are to be sent. There are two main ways in which addresses are encoded so that they can be communicated from receiver to spender: as a text string or as a QR code.

To encode an address as a text string, we take the bits of the key and convert it from a binary number to a base 58 number. Then we use a set of 58 characters to encode each digits as a character; this is called base58 notation. Why 58? Because that's the number we get when we include the upper case letters, lower case letters, as well as digits as characters, but leave out a few that might be confusing or might look like another character. For example, capital letter 'O' and zero are both taken out because they look too much alike. This allows encoded addresses to be read out over the phone or read from printed paper and typed in, should that be necessary. Ideally such manual methods of communicating addresses can be avoided through methods such as QR codes, which we now discuss.

`1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa`

The address that received the very first Bitcoin block reward in the genesis block, base58 encoded.

Figure 4.1: **a QR code representing an actual Bitcoin address.** Feel free to send us some bitcoins.

The second method for encoding a Bitcoin address is as a QR code, a simple kind of 2-dimensional barcode. The advantage of a QR code is that you can take a picture of it with a smartphone and wallet software can automatically turn the barcode into the a sequence of bits that represents the corresponding Bitcoin address. This is useful in a store, for example: the check-out system might display a QR code and you can pay with your phone by scanning the code and sending coins to that address. It is also useful for phone-to-phone transfers.

4.2 Hot and Cold Storage

As we just saw, storing bitcoins on your computer is like carrying money around in your wallet or your purse. This is called “hot storage”. It’s convenient but also somewhat risky. On the other hand, “cold storage” is offline. It's locked away somewhere. It's not connected to the internet, and it's archival. So it’s safer and more secure, but of course, not as convenient. This is similar to how you carry some money around on your person, but put your life's savings somewhere safer.

To have separate hot and cold storage, obviously you need to have separate secret keys for each — otherwise the coins in cold storage would be vulnerable if the hot storage is compromised. You’ll want to move coins back and forth between the hot side and the cold side, so each side will need to know the other’s addresses, or public keys.

Cold storage is not online, and so the hot storage and the cold storage won't be able to connect to each other across any network. But the good news is that cold storage doesn’t have to be online to receive coins — since the hot storage knows the cold storage addresses, it can send coins to cold storage at any time. At any time if the amount of money in your hot wallet becomes uncomfortably large, you can transfer a chunk of it over to cold storage, without putting your cold storage at risk by connecting to the network. Next time the cold storage connects it will be able to receive from the block chain information about those transfers to it and then the cold storage will be able to do what it wants with those coins.

But there’s a little problem when it comes to managing cold storage addresses. On the one hand, as we saw earlier, for privacy and other reasons we want to be able to receive each coin at a separate address with different secret keys. So whenever we transfer a coin from the hot side to the cold side we'd like to use a fresh cold address for that purpose. But because the cold side is not online we have to have some way for the hot side to find out about those addresses.

The blunt solution is for the cold side generate a big batch of addresses all at once and send those over for the hot side to use them up one by one. The drawback is that we have to periodically reconnect the cold side in order to transfer more addresses.

Hierarchical wallets. A more effective solution is to use a hierarchical wallet. It allows the cold side to use an essentially unbounded number of addresses and the hot side to know about these addresses, but with only a short, one-time communication between the two sides. But it requires a little bit of cryptographic trickery.

To review, previously when we talked about key generation and digital signatures back in chapter 1, we looked at a function called generateKeys that generates a public key (which acts as an address) and a secret key. In a hierarchical wallet, key generation works differently. Instead of generating a single address we generate what we'll call address generation info, and rather than a private key we generate what we'll call private key generation info. Given the address generation info, we can generate a sequence of addresses: we apply an address generation function that takes as input the address generation info and any integer *i* and generate the *i*'th address in the sequence. Similarly we can generate a sequence of private keys using the private key generation info.

The cryptographic magic that makes this useful is that the for every *i*, the *i*'th address and *i*'th secret key "match up" — that is, the *i*'th secret key controls, and can be used to spend, bitcoins from the *i*'th address just as if the pair were generated the old fashioned way. So it's as if we have a sequence of regular key pairs.

The other important cryptographic property here is security: the address generation info doesn't leak any information about the private keys. That means that it's safe to give the address generation info to anybody, and so that anybody can be enabled to generate the 'i'th key.

Now, not all digital signature schemes that exist can be modified to support hierarchical key generation. Some can and some can't, but the good news is that the digital signature scheme used by Bitcoin, ECDSA, does support hierarchical key generation, allowing this trick. That is, the cold side generates an arbitrarily many keys and the hot side generates the corresponding addresses.

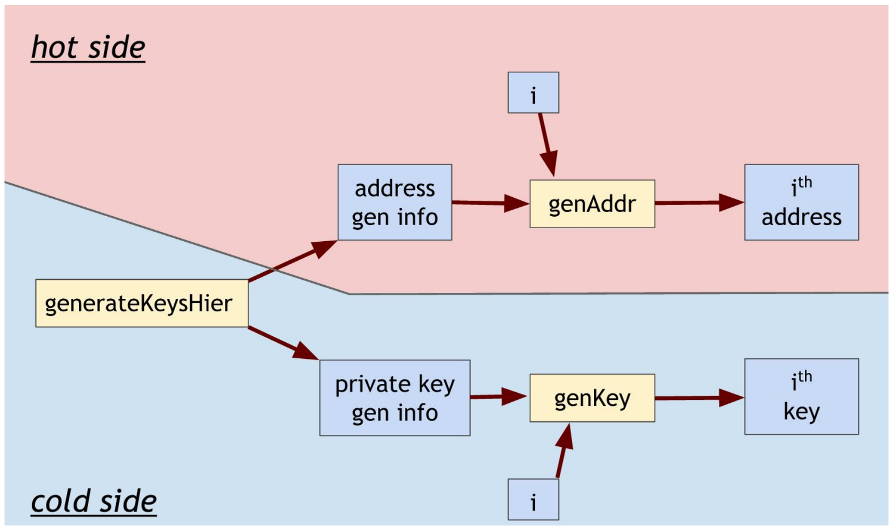


Figure 4.2: Schema of a hierarchical wallet. The cold side creates and saves private key generation info and address generation info. It does a one-time transfer of the latter to the hot side. The hot side generates a new address sequentially every time it wants to send coins to the cold side. When the cold side reconnects, it generates addresses sequentially and checks the block chain for transfers to those addresses until it reaches an address that hasn't received any coins. It can also generate private keys sequentially if it wants to send some coins back to the hot side or spend them some other way.

Now let’s talk about the different ways in which cold information — whether one or more keys, or key-generation info — can be stored. The first way is to store it in some kind of device and put that device in a safe. It might be a laptop computer, a mobile phone or tablet, or a thumb drive. The important thing is to turn the device off and lock it up, so that if somebody wants to steal it they have to break into the locked storage.

Brain wallet. The second method we can use is called a brain wallet. This is a way to control access to bitcoins using nothing but a secret passphrase. This avoids the need for hard drives, paper, or any other long-term storage mechanism. This property can be particularly useful in situations where you have poor physical security, perhaps when you’re traveling internationally.

The key trick behind a brain wallet is to have a predictable algorithm for turning a passphrase into a public and private key. For example, you could hash the passphrase with a suitable hash function to derive the private key, and given the private key, the public key can be derived in a standard way. Further, combining this with the hierarchical wallet technique we saw earlier, a we can generate an entire sequence of addresses and private keys from a passphrase, thus enabling a complete wallet.

However, an adversary can also obtain all private keys in a brain wallet if they can guess the passphrase. As always in computer security, we must assume that the adversary knows the procedure you used to generate keys, and only your passphrase provides security. So the adversary can try various passphrases and generate addresses using them; if he finds any unspent transactions on the block chain at any of those addresses, he can immediately transfer them to himself. The adversary may never know (or care) who the coins belonged to and the attack doesn’t require breaking into any machines. Guessing brain wallet passphrases is not directed toward specific users, and further, leaves no trace.

Furthermore, unlike the task of guessing your email password which can be *rate-limited* by your email server (called *online guessing*), with brain wallets the attacker can download the list of addresses with unredeemed coins and try as many potential passphrases as they have the computational capacity to check. Note that the attacker doesn’t need to know which addresses correspond to brain wallets. This is called *offline guessing* or *password cracking*. It is much more challenging to come up with passphrases that are easy to memorize and yet won’t be vulnerable to guessing in this manner. One secure way to generate a passphrase is to have an automatic procedure for picking a random 80-bit number and turning that number into a passphrase in such a way that different numbers result in different passphrases.

In practice, it is also wise to use a deliberately slow function to derive the private key from the passphrase (referred to as *key stretching*) to ensure it takes as long as possible for the attacker to try all possibilities. The basic approach is to take a fast cryptographic hash function like SHA-256 and compute perhaps 2^{20} iterations of it, multiplying the attacker’s workload by a factor of 2^{20}. Of course, if it is too slow it will start to become annoying to the user as their device must re-compute this function any time they want to spend coins from their brain wallet.

If a brain wallet passphrase is inaccessible — say it’s been forgotten, hasn’t been written down, and can’t be guessed — then the coins are lost forever.

Paper wallet. The third option is what's called a paper wallet. We can print the key material to paper and then put that paper into a safe or secure place. Obviously, the security of this method is just as good or bad as the physical security of the paper that we're using. Typical paper wallets encode both the public and private key in two ways: as a 2D barcode and in base 58 notation. Just like with a brain wallet, storing a small amount of key material is sufficient to re-create a wallet.

Tamper-resistant device. The fourth way that we can store offline information is to put it in some kind of tamper-resistant device. Either we put the key into the device or the device generates the key; either way, the device is designed so that there's no way it will output or divulge the key. The device instead signs statements with the key, and does so when we, say, press a button or give it some kind of password. One advantage is that if the device is lost or stolen we'll know it, and the only way the key can be stolen is if the device is stolen. This is different from storing your key on a laptop.

In general, people might use a combination of four of these methods in order to secure their keys. For hot storage, and especially for hot storage holding large amounts of bitcoins, people are willing to work pretty hard and come up with novel security schemes in order to protect them, and we'll talk a little bit about one of those more advanced schemes in the next section.

4.3 Splitting and Sharing Keys

Up to now we've looked at different ways of storing and managing the secret keys that control bitcoins, but we've always put a key in a single place — whether locked in a safe, or in software, or on paper. This leaves us with a single point of failure. If something goes wrong with that single storage place then we're in trouble. We could create and store backups of the key material, but while this decreases the risk of the key getting lost or corrupted (availability), it *increases* the risk of theft (security). This trade-off seems fundamental. Can we take a piece of data and store it in such a way that availability and security increase at the same time? Remarkably, the answer is yes, and it is once again a trick that uses cryptography, called *secret sharing*.

Here’s the idea: we want to divide our secret key into some number N of pieces. We want to do it in such a way that if we're given any K of those pieces then we'll be able to reconstruct the original secret, but if we're given fewer than K pieces then we won't be able to learn anything about the original secret.

Given this stringent requirement, simply “cutting up” the secret into pieces won’t work because even a single piece gives some information about the secret. We need something cleverer. And since we’re not cutting up the secret, we’ll call the individual components “shares” instead of pieces.

Let’s say we have N=2 and K=2. That means we're generating 2 shares based on the secret, and we need both shares to be able to reconstruct the secret. Let’s call our secret S, which is just a big (say 128-bit) number. We could generate a 128-bit random number R and make the two shares be R and S ⊕ R. (⊕ represents bitwise XOR). Essentially, we’ve “encrypted” S with a one-time pad, and we store the key (R) and the ciphertext (S ⊕ R) in separate places. Neither the key nor the ciphertext by itself tells us anything about the secret. But given the two shares, we simply XOR them together to reconstruct the secret.

This trick works as long as N and K are the same — we’d just need to generate N-1 different random numbers for the first N-1 shares, and the final share would be the secret XOR’d with all other N-1 shares. But if N is more than K, this doesn’t work any more, and we need some algebra.

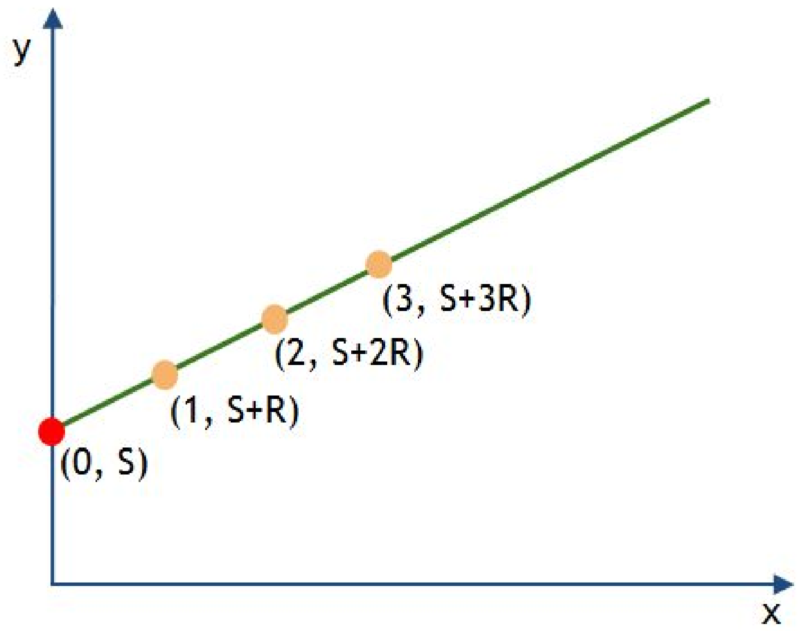


Figure 4.3: Geometric illustration of 2-out-of-N secret sharing. S represents the secret, encoded as a (large) integer. The green line has a slope chosen at random. The orange points (specifically, their Y-coordinates S+R, S+2R, ...) correspond to shares. Any two orange points are sufficient to reconstruct the red point, and hence the secret. All arithmetic is done modulo a large prime number.

Take a look at Figure 4.3. What we’ve done here is to first generate the point (0, S) on the Y-axis, and then drawn a line with a random slope through that point. Next we generate a bunch points on that line, as many as we want. It turns out that this is a secret sharing of S with N being the number of points we generated and K=2.

Why does this work? First, if you’re given two of the points generated, you can draw a line through them and see where it meets the Y-axis. That would give you S. On the other hand, if you’re given only a single point, it tells you nothing about S, because the slope of the line is random. Every line through your point is equally likely, and they would all intersect the Y-axis at different points.

There's only one other subtlety: to make the math work out, we have to do all our arithmetic modulo a large prime number P. It doesn't need to be secret or anything, just really big. And the secret S has to be between 0 and P-1, inclusive. So when we say we generate points on the line, what we mean is that we generate a random value R, also between 0 and P-1, and the points we generate are

x=1, y=(S+R) mod P
x=2, y=(S+2R) mod P
x=3, y=(S+3R) mod P

and so on. The secret corresponds to the point x=0, y=(S+0*R) mod P, which is just x=0, y=S.

What we've seen is a way to do secret sharing with K=2 and any value of N. This is already pretty good — if N=4, say, you can divide your secret key into 4 shares and put them on 4 different devices so that if someone steals any one of those devices, they learn nothing about your key. On the other hand, even if two of those devices are destroyed in a fire, you can reconstruct the key using the other two. So as promised, we've increased both availability and security.

But we can do better: we can do secret sharing with any N and K as long as K is no more than N. To see how, let's go back to the figure. The reason we used a line instead of some other shape is that a line, algebraically speaking, is a polynomial of degree 1. That means that to reconstruct a line we need two points and no fewer than two. If we wanted K=3, we would have used a parabola, which is a quadratic polynomial, or a polynomial of degree 2. Exactly three points are needed to construct a quadratic function. We can use the table below to understand what's going on.

Equation	**Degree**	**Shape**	**Random parameters**	**Number of points (K) needed to recover S**
$(S + RX) \bmod P$	1	Line	R	2
$(S + R_1X + R_2X^2) \bmod P$	2	Parabola	R_1, R_2	3
$(S + R_1X + R_2X^2 + R_3X^3) \bmod P$	3	Cubic	R_1, R_2, R_3	4

Table 4.1: The math behind secret sharing. Representing a secret via a series of points on a random polynomial curve of degree K-1 allows the secret to be reconstructed if, and only if, at least K of the points ("shares") are available.

There is a formula called Lagrange interpolation that allows you to reconstruct a polynomial of degree K-1 from any K points on its curve. It's an algebraic version (and a generalization) of the geometric intuition of drawing a straight line through two points with a ruler. As a result of all this, we have a way to store any secret as N shares such that we're safe even if an adversary learns up to K-1 of them, and at the same time we can tolerate the loss of up to N-K of them.

None of this is specific to Bitcoin, by the way. You can secret-share your passwords right now and give shares to your friends or put them on different devices. But no one really does this with secrets like passwords. Convenience is one reason; another is that there are other security mechanisms available for important online accounts, such as two-factor security using SMS verification. But with Bitcoin, if you're storing your keys locally, you don't have those other security options. There's no way to make the control of a Bitcoin address dependent on receipt of an SMS message. The situation is different with online wallets, which we'll look at in the next section. But not too different — it just shifts the problem to a different place. After all, the online wallet provider will need some way to avoid a single point of failure when storing *their* keys.

Threshold cryptography. But there's still a problem with secret sharing: if we take a key and we split it up in this way and we then want to go back and use the key to sign something, we still need to bring the shares together and recalculate the initial secret in order to be able to sign with that key. The point where we bring all the shares together is still a single point of vulnerability where an adversary might be able to steal the key.

Cryptography can solve this problem as well: if the shares are stored in different devices, there's a way to produce Bitcoin signatures in a decentralized fashion without ever reconstructing the private key on any single device. This is called a "threshold signature." The best use-case is a wallet with two-factor security, which corresponds to the case N=2 and K=2. Say you've configured your wallet to split its key material between your desktop and your phone. Then you might initiate a payment on your desktop, which would create a partial signature and send it to your phone. Your phone would then alert you with the payment details — recipient, amount, etc. — and request your confirmation. If the details check out, you'd confirm, and your phone would complete the signature using its share of the private key and broadcast the transaction to the block chain. If there were malware on your desktop that tried to steal your bitcoins, it might initiate a transaction that sent the funds to the hacker's address, but then you'd get an alert on your phone for a transaction you didn't authorize, and you'd know something was up. The mathematical details behind threshold signatures are complex and we won't discuss them here.

Multi-signatures. There's an entirely different option for avoiding a single point of failure: multi-signatures, which we saw earlier in Chapter 3. Instead of taking a single key and splitting it, Bitcoin script directly allows you to stipulate that control over an address be split between different keys. These keys can then be stored in different locations and the signatures produced separately. Of course, the completed, signed transaction will be constructed on some device, but even if the adversary controls this device, all that he can do is to prevent it from being broadcast to the network. He can't produce valid multi-signatures of some other transaction without the involvement of the other devices.

As an example, suppose that Andrew, Arvind, Ed, Joseph, and Steven, the authors of this book, are co-founders of a company — perhaps we started it with the copious royalties from the sale of this free book — and the company has a lot of bitcoins. We might use multi-sig to protect our large store of bitcoins. Each of the five of us will generate a key pair, and we’ll protect our cold storage using 3-out-of-5 multi-sig, which means that three of us must sign to create a valid transaction.

As a result, we know that we're relatively secure if the five of us keep our keys separately and secure them differently. An adversary would have to compromise three out of the five keys. If one or even two of us go rogue, they can't steal the company’s coins because you need at least three keys to do that. At the same time, if one of us loses our key or gets run over by a bus and our brain wallet is lost, the others can still get the coins back and transfer them over to a new address and re-secure the keys. In other words, multi-sig helps you to manage large amounts of cold-stored coins in a way that's relatively secure and requires action by multiple people before anything drastic happens.

Sidebar. Threshold signatures are a cryptographic technique to take a single key, split it into shares, store them separately, and sign transactions without reconstructing the key. Multi-signatures are a feature of Bitcoin script by which you can specify that control of an address is split between multiple independent keys. While there are some differences between them, they both increase security by avoiding single points of failure.

4.4 Online Wallets and Exchanges

So far we've talked about ways in which you can store and manage your bitcoins itself. Now we'll talk about ways you can use other people’s services to help you do that. The first thing you could do is use an online wallet.

Online wallets. An online wallet is kind of like a local wallet that you might manage yourself, except the information is stored in the cloud, and you access it using a web interface on your computer or using an app on your smartphone. Some online wallet services that are popular in early 2015 are Coinbase and blockchain.info.

What’s crucial from the point of view of security is that the site delivers the code that runs on your browser or the app, and it also stores your keys. At least it will have the ability to access your keys. Ideally, the site will encrypt those keys under a password that only you know, but of course you have to trust them to do that. You have to trust their code to not leak your keys or your password.

An online wallet has certain trade offs to doing things yourself. A big advantage is that it's convenient. You don't have to install anything on your computer in order to be able to use an online wallet in your browser. On your phone you maybe just have to install an app once, and it won’t need to download the block chain. It will work across multiple devices — you can have a single wallet that you access on your desktop and on your phone and it will just work because the real wallet lives in the cloud.

On the other hand, there are security worries. If the site or the people who operate the site turn out to be malicious or are compromised somehow, your bitcoins are in trouble. The site supplies the code that has its grubby fingers on your bitcoins, and things can go wrong if there's a compromise or a malice at the service provider.

Ideally, the site or the service is run by security professionals who are better trained, or perhaps more diligent than you in maintaining security. So you might hope that they do a better job and that your coins are actually more secure than if you stored them yourself. But at the end of day, you have to trust them and you have to rely on them not being compromised.

Bitcoin exchanges. To understand Bitcoin exchanges, let's first talk about how banks or bank like services operate in the traditional economy. You give the bank some money — a deposit — and the bank promises to give you back that money later. Of course, crucially, the bank doesn't actually just take your money and put it in a box in the back room. All the bank does is promise that if you show up for the money they'll give it back. The bank will typically take the money and put it somewhere else, that is, invest it. The bank will probably keep some money around in reserve in order to make sure that they can pay out the demand for withdrawals that they'll face on a typical day, or maybe even an unusual day. Many banks typically use something called ***fractional reserve*** where they keep a certain fraction of all the demand deposits on reserve just in case.

Now, Bitcoin exchanges are businesses that at least from the user interface standpoint function in a similar way to banks. They accept deposits of bitcoins and will, just like a bank, promise to give them back on demand later. You can also transfer fiat currency — traditional currency like dollars and euros — into an exchange by doing a transfer from your bank account. The exchange promises to pay back either or both types of currency on demand. The exchange lets you do various banking-like things. You can make and receive Bitcoin payments. That is, you can direct the exchange to pay out some bitcoins to a particular party, or you can ask someone else to deposit funds into the particular exchange on your behalf — put into your account. They also let you exchange bitcoins for fiat currency or vice versa. Typically they do this by finding some customer who wants to buy bitcoins with dollars and some other customer who wants to sell bitcoins for dollars, and match them up. In other words, they try to find customers willing to take opposite positions in a transaction. If there’s a mutually acceptable price, they will consummate that transaction.

Suppose my account at some exchange holds 5000 dollars and three bitcoins and I use the exchange, I put in an order to buy 2 bitcoins for 580 dollars each, and the exchange finds someone who is willing to take the other side of that transaction and the transaction happens. Now I have five bitcoins in my account instead of three, and 3840 dollars instead of 5000.

The important thing to note here is that when this transaction happened involving me and another customer of the same exchange, no transaction actually happened on the Bitcoin block chain. The exchange doesn’t need to go to the block chain in order to transfer bitcoins or dollars from account to another. All that happens in this transaction is that the exchange is now making a different promise to me then they were making before. Before they said, “we'll give you 5000 USD and 3 BTC” and now they're saying “we'll give you 3840 USD and 5 BTC.” It's just a change in their promise — no actual movement of money through the dollar economy or through the block chain. Of course, the other person has had their promises to them change in the opposite way.

There are pros and cons to using exchanges. One of the big pros is that exchanges help to connect the Bitcoin economy and the flows of bitcoins with the fiat currency economy so that it's easy to transfer value back and forth. If I have dollars and bitcoins in my account I can trade back and forth between them pretty easily, and that's really helpful.

The con is risk. You have the same kind of risk that you face with banks, and those risks fall into three categories.

Three types of risks. The first risk is the risk of a ***bank run***. A run is what happens when a bunch of people show up all at once and want their money back. Since the bank maintains only fractional reserves, it might be unable to cope with the simultaneous withdrawals. The danger is a kind of panic behavior where once the rumor starts to get around that a bank or exchange might be in trouble and they might be getting close to not honoring withdrawals, then people stampede in to try to withdraw their money ahead of the crowd, and you get a kind of avalanche.

The second risk is that the owners of the banks might just be crooks running a Ponzi scheme. This is a scheme where someone gets people to give them money in exchange for profits in the future, but then actually takes their money and uses it to pay out the profits to people who bought previously. Such a scheme is doomed to eventually fail and lose a lot of people a lot of money. Bernie Madoff most famously pulled this off in recent memory.

The third risk is that of a hack, the risk that someone — perhaps even an employee of the exchange — will manage to penetrate the security of the exchange. Since exchanges store key information that controls large amounts of bitcoins, they need to be really careful about their software security and their procedures — how they manage their cold and hot storage and all of that. If something goes wrong, your money could get stolen from the exchange.

All of these things have happened. We have seen exchanges that failed due to the equivalent of a bank run. We've seen exchanges fail due to the operators of the exchange being crooks, and we've seen exchanges that fail due to break-ins. In fact, the statistics are not encouraging. A study in 2013 found that 18 of 40 Bitcoin exchanges had ended up closing due to some failure or some inability to pay out the money that the exchange had promised to pay out.

The most famous example of this of course is Mt. Gox. Mt. Gox was at one time the largest Bitcoin exchange, and it eventually found itself insolvent, unable to pay out the money that it owed. Mt. Gox was a Japanese company and it ended up declaring bankruptcy and leaving a lot of people wondering where their money had gone. Right now the bankruptcy of Mt. Gox is tangled up in the Japanese and American courts, and it's going to be a while before we know exactly where the money went. The one thing we know is that there's a lot of it and Mt. Gox doesn't have it anymore. So this is a cautionary tale about the use of exchanges.

Connecting this back to banks, we don't see a 45% failure rate for banks in most developed countries, and that’s partly due to regulation. Governments regulate traditional banks in various ways.

Bank regulation. The first thing that governments do is they often impose a minimum reserve requirement. In the U.S., the fraction of demand deposits that banks are required to have in liquid form is typically 3-10%, so that it can deal with a surge of withdrawals if that happens. Second, governments often regulate the types of investments and money management methods that banks can use. The goal is to ensure that the banks’ assets are invested in places that are relatively low risk, because those are really the assets of the depositors in some sense.

Now, in exchange for these forms of regulation governments typically do things to help banks or help their depositors. First, governments will issue deposit insurance. That is, the government promises depositors that if a bank that follows these rules goes under, the government will make good on at least part of those deposits. Governments also sometimes act as a “lender of last resort.” If a bank gets itself into a tough spot, but it's basically solvent, the government may step in and loan the bank money to tide it over until it can move money around as necessary to get itself out of the woods.

So, traditional banks are regulated in this way. Bitcoin exchanges are not. The question of whether or how Bitcoin exchanges or other Bitcoin business should be regulated is a topic that we will come back to in chapter 7.

Proof of reserve. A Bitcoin exchange or someone else who holds bitcoins can use a cryptographic trick called a proof of reserve to give customers some comfort about the money that they deposited. The goal is for the exchange or business holding bitcoins to prove that it has a fractional reserve — that they retain control of perhaps 25% or maybe even 100% of the deposits that people have made.

We can break the proof-of-reserve problem into two pieces. The first is to prove how much reserve you’re holding — that's the relatively easy part. The company simply publishes a valid payment-to-self transaction of the claimed reserve amount. That is, if they claim to have 100,000 bitcoins, they create a transaction in which they pay 100,000 bitcoins to themselves and show that that transaction is valid. Then they sign a challenge string — a random string of bits generated by some impartial party — with the same private key that was used to sign the payment-to-self transaction. This proves that someone who knew that private key participated in the proof of reserve.

We should note two caveats. Strictly speaking, that's not a proof that the party that's claiming to own the reserve owns it, but only that whoever does own those 100,000 bitcoins is willing to cooperate in this process. Nonetheless, this looks like a proof that somebody controls or knows someone who controls the given amount of money. Also, note that you could always under-claim: the organization might have 150,000 bitcoins but choose to make a payment-to-self of only 100,000. So this proof of reserve doesn't prove that this all you have, but it proves that you have at least that much.

Proof of liabilities. The second piece is to prove how many demand deposits you hold, which is the hard part. If you can prove your reserves and you demand deposits then anyone can simply divide those two numbers and that's what your fractional reserve is. We'll present a scheme that allows you to *over-claim* but not under-claim your demand deposits. So if you can prove that your reserves are at least a certain amount and your liabilities are at most a certain amount, taken together, you've proved a lower bound on your fractional reserve.

If you didn't care at all about the privacy of your users, you could simply publish your records — specifically, the username and amount of every customer with a demand deposit. Now anyone can calculate your total liabilities, and if you omitted any customer or lied about the value of their deposit, you run the risk that that customer will expose you. You could make up fake users, but you can only increase the value of your claimed total liabilities this way. So as long as there aren't customer complaints, this lets you prove a lower bound on your deposits. The trick, of course, is to do all this while respecting the privacy of your users.

To do this we'll use Merkle trees, which we saw in chapter 1. Recall that a merkle tree is a binary tree that's built with hash pointers so that each of the pointers not only says where we can get a piece of information, but also what the cryptographic hash of that information is. The exchange executes the proof by constructing a Merkle tree in which each leaf corresponds to a user, and publishing its root hash. Similar to the naive protocol above, it's each user's responsibility to ensure that they are included in the tree. In addition, there's a way for users to collectively check the claimed total of deposits. Let's delve into detail now.

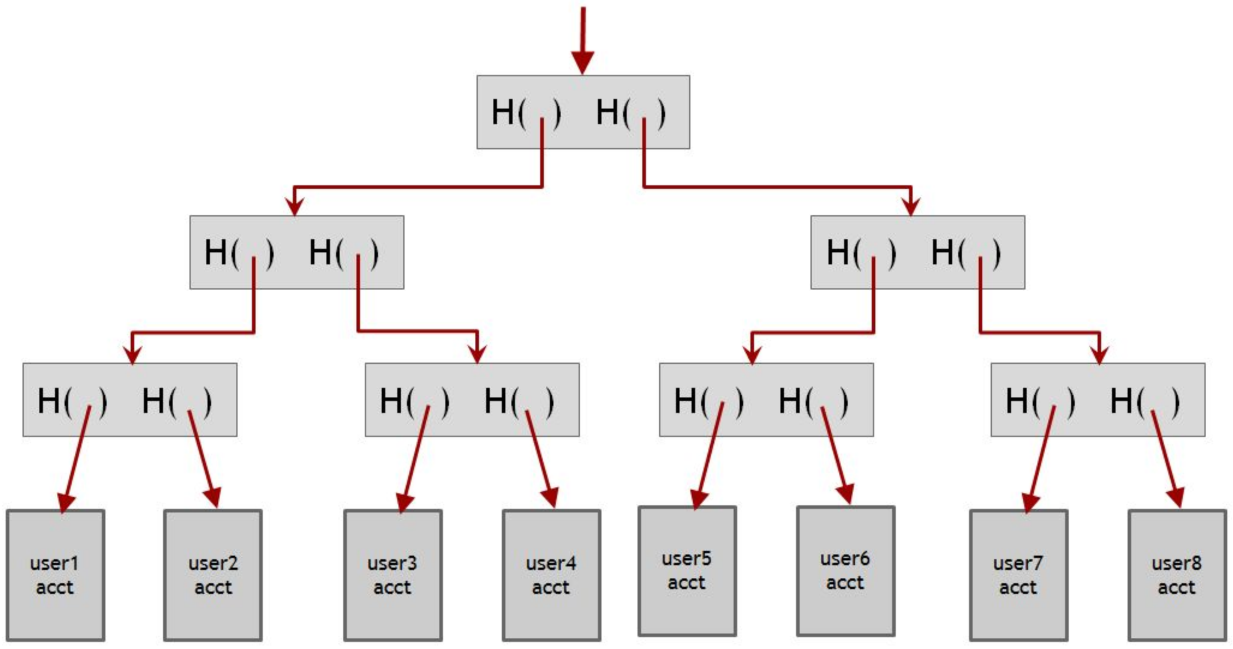


Figure 4.4: Proof of liabilities. The exchange publishes the root of a Merkle tree that contains all users at the leaves, including deposit amounts. Any user can request a proof of inclusion in the tree, and verify that the deposit sums are propagated correctly to the root of the tree.

Now, we're going to add to each one of these hash pointers another field, or attribute. This attribute is a number that represents the total monetary value in bitcoins of all of all deposits that are in the sub-tree underneath that hash pointer in the tree. For this to be true, the value corresponding to each hash pointer should be the sum of the values of the two hash pointers beneath it.

The exchange constructs this tree, cryptographically signs the root pointer along with the root attribute value, and publishes it. The root value is of course the total liabilities, the number we're interested in. The exchange is making the claim that all users are represented in the leaves of the tree, their deposit values are represented correctly, and that the values are propagated correctly up the tree so that the root value is the sum of all users' deposit amounts.

Now each customer can go to the organization and ask for a proof of correct inclusion. The exchange must then show the customer the partial tree from that user's leaf up to the root, as shown in Figure 4.5. The customer then verifies that:

1. The root hash pointer and root value are the same as what the exchange signed and published.
2. The hash pointers are consistent all the way down, that is, each hash value is indeed the cryptographic hash of the node it points to.
3. The leaf contains the correct user account info (say, username/user ID, and deposit amount).
4. Each value is the sum of the values of the two values beneath it.

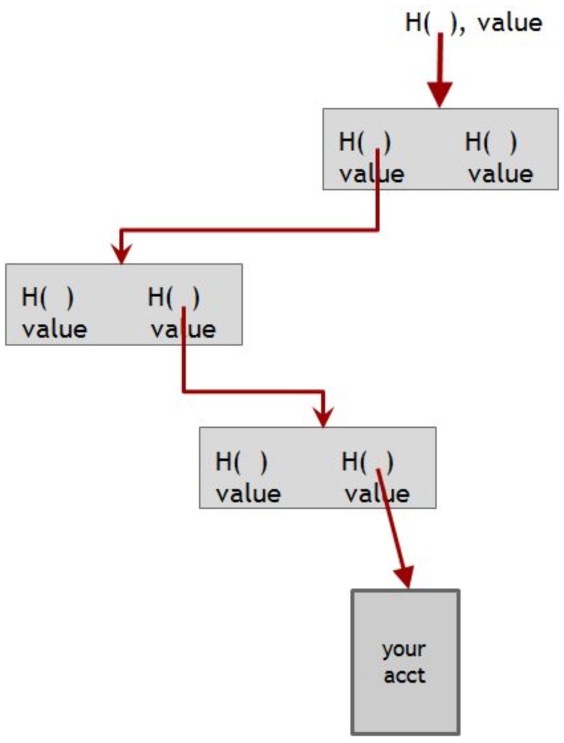


Figure 4.5: Proof of inclusion in a Merkle tree. The leaf node is revealed, as well as the siblings of the nodes on the path from the leaf to the root.

The good news is that if every customer does this, then every branch of this tree will get explored, and someone will verify that for every hash pointer, its associated value equals the sum of the values of its two children. Crucially, the exchange cannot present different values in any part of the tree to different customers. That’s because doing so would either imply the ability find a hash collision, or presenting different root values to different customers, which we assume is impossible.

Let’s recap. First the exchange proves that they have at least X amount of reserve currency by doing a self transaction of X amount. Then they prove that their customers have at most an amount Y deposited. This shows that their reserve fraction is at least X/Y. What that means is that if a Bitcoin exchange wants to prove that they hold 25% reserves on all deposits — or 100% — they can do that in a way that's independently verifiable by anybody, and no central regulator is required.

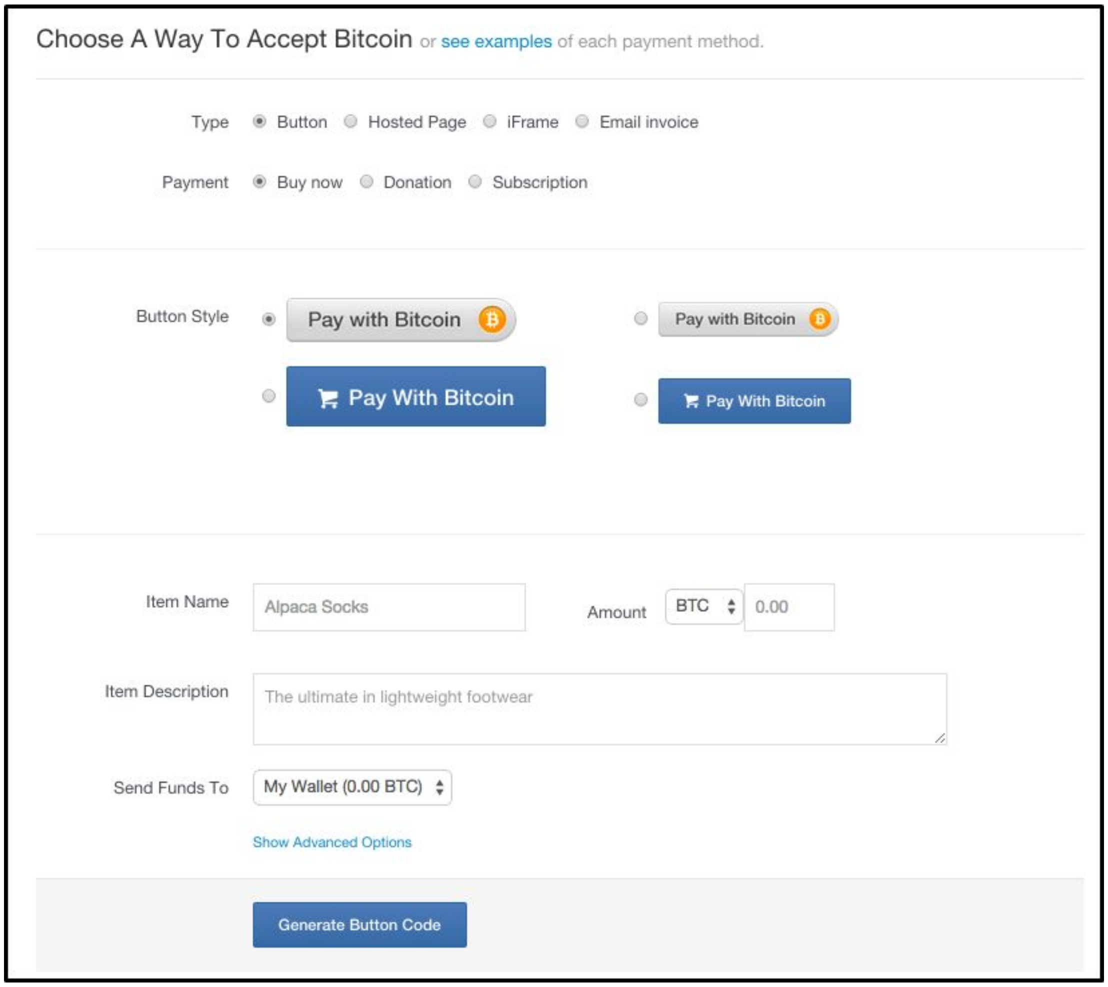
This is one aspect of regulation that Bitcoin exchanges can prove voluntarily, but other aspects of regulation are harder to guarantee, as we'll see in Chapter 7.

4.5 Payment Services

So far we've talked about how you can store and manage your bitcoins. Now let’s consider how a merchant — whether an online merchant or a local retail merchant — can accept payments in bitcoins in a practical way. Merchants generally support Bitcoin payments because their customers want to be able to pay with bitcoins. The merchant may not want to hold on to bitcoins, but simply receive dollars or whatever is the local fiat currency at the end of the day. They want an easy way to do this without worrying too much about technology, changing their website or building some type of point of sale technology.

The merchant also wants low risk. There are various possible risks: using new technology may cause their website to go down, costing them money. There’s the security risk of handling bitcoins — someone might break into their hot wallet or some employee will make off with their bitcoins. Finally there’s the exchange rate risk: the value of bitcoins in dollars might fluctuate from time to time. The merchant who might want to sell a pizza for twelve dollars wants to know that they're going to get twelve dollars or something close to it, and that the value of the bitcoins that they receive in exchange for that pizza won't drop drastically before they can exchange those bitcoins for dollars.

Payment services exist to allow both the customer and the merchant to both get what they want, bridging the gap between these different desires.



Choose A Way To Accept Bitcoin or see examples of each payment method.

Type	[✓ Button] Hosted Page iFrame Email invoice
Payment	[✓ Buy now] Donation Subscription
Button Style	[✓ Pay with Bitcoin]
Item Name	Alpaca Socks
Amount	BTC 0.00
Item Description	The ultimate in lightweight footwear
Send Funds To	My Wallet (0.00 BTC)

Figure 4.6: Example payment service interface for generating a pay-with-Bitcoin button. A merchant can use this interface to generate a HTML snippet to embed on their website.

The process of receiving Bitcoin payments through a payment service might look like this to the merchant:

1. The merchant goes to payment service website fills out a form describing the item, price, and presentation of the payment widget, and so on. Figure 4.6 shows an illustrative example of a form from Coinbase.
2. The payment service generates HTML code that the merchant can drop into their website.
3. When the customer clicks the payment button, various things happen in the background and eventually the merchant gets a confirmation saying, “a payment was made by customer ID [customer-id] for item [item-id] in amount [value].”

While this manual process makes sense for a small site selling one or two items, or a site wishing to receive donations, copy-pasting HTML code for thousands of items is of course infeasible. So payment services also provide programmatic interfaces for adding a payment button to dynamically generated web pages.

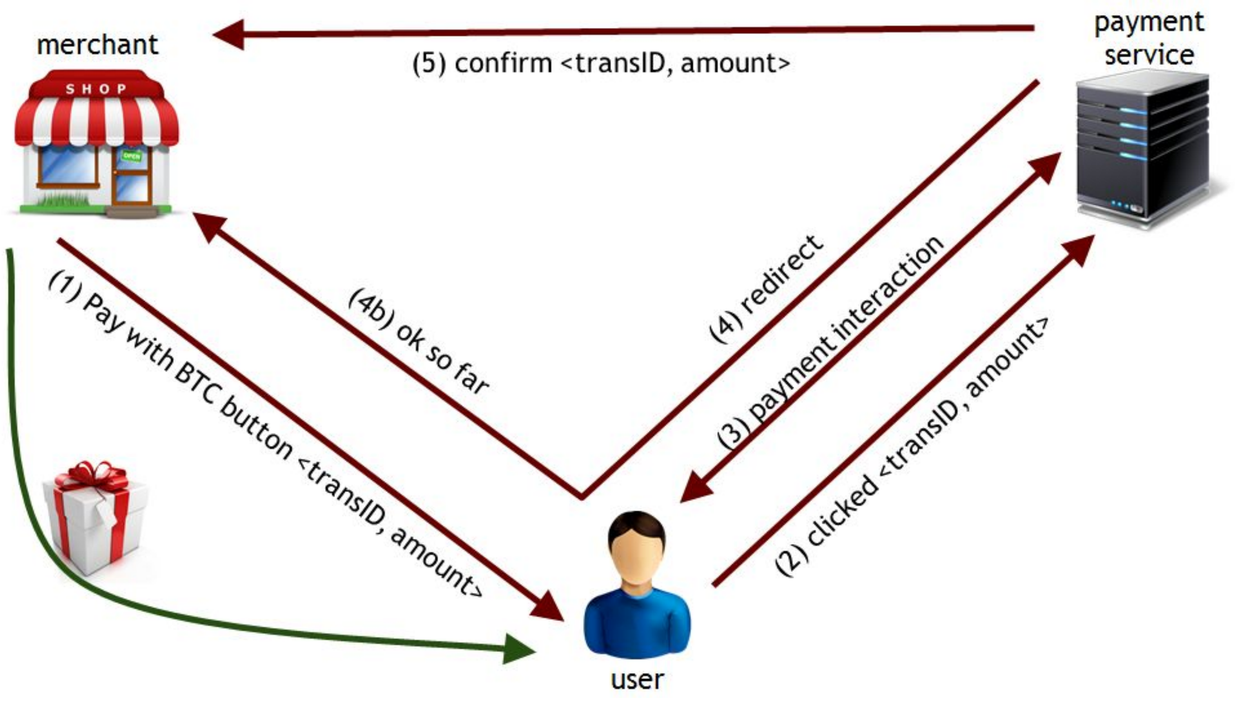

merchant
payment service
(5) confirm <transID, amount>
(1) Pay with BTC button <transID, amount>
(4b) ok so far
(4) redirect
(3) payment interaction
(2) clicked <transID, amount>
user

Figure 4.7: Payment process involving a user, merchant, and payment service.

Now let's look at the payment process in more detail to see what happens when the customer makes a purchase with Bitcoin. The steps below are illustrated in Figure 4.7.

1. The user picks out an item to buy on the merchant website, and when it comes times to pay, the merchant will deliver a webpage which will contain the Pay with Bitcoin button, which is the HTML snippet provided by the payment service. The page will also contain a transaction ID — which is an identifier that's meaningful to the merchant and allows them to locate a record in their own accounting system — along with an amount the merchant wants to be paid.
2. If the user wants to pay with bitcoins, they will click that button. That will trigger an HTTPS request to the payment service says that button was clicked, and passing on the identity of the merchant, the merchant's transaction ID, and the amount.
3. Now the payment service knows that this customer — whoever they are — wants to pay a certain amount of bitcoins, and so the payment service will pop up some kind of a box, or initiate some kind of an interaction with the user. This gives the user information about how to pay, and the user will then initiate a bitcoin transfer to the payment service through their preferred wallet.
4. Once the user has created the payment, the payment service will redirect the browser to the merchant, passing on the message from the payment service that it looks okay so far. This might mean, for example, that the payment service has observed the transaction broadcast to the peer-to-peer network, but the transaction hasn't received enough (or any) confirmations so far. This completes the payment as far as the user is concerned, with the merchant's shipment of goods pending a final confirmation from the payment service.

5. The payment service later directly sends a confirmation to the merchant containing the transaction ID and amount. By doing this the payment service tells the merchant that the service owes the merchant money at the end of the day. The merchant then ships the goods to the user.

The final step is the one where the payment service actually sends money to the merchant, in dollars or some fiat currency, via a deposit to the merchant's bank account. This happens at the end of fixed settlement periods, perhaps once a day, rather than once for each purchase. The payment service keeps a small percentage as a fee; that's how they make their revenue. Some of these details might vary depending on the payment service, but this is the general scheme of things.

To recap, at the end of this process the customer pays bitcoins and the merchant gets dollars, minus a small percentage, and everyone is happy. Recall that the merchant wants to sell items for a particular number of dollars or whatever is the local fiat currency. The payment service handles everything else — receiving bitcoins from customers and making deposits at the end of the day.

Crucially, the payment service absorbs all of the risk. It absorbs the security risk, so it needs to have good security procedures to manage its bitcoins. It absorbs the exchange rate risk because it's receiving bitcoins and paying out dollars. If the price of dollars against bitcoins fluctuates wildly, the payment service might lose money. But then if it fluctuates wildly in the other direction the service might earn money, but it's a risk. Absorbing it is part of the payment service's business.

Note that the payment service probably operates at a large scale, so is receives large numbers of bitcoins and pays out large numbers of dollars. it will have a constant need to exchange the bitcoins it's receiving for more dollars so that it can keep the cycle going. Therefore a payment service has to be an active participant in the exchange markets that link together fiat currencies and the Bitcoin economy. So the service needs to worry about not just what the exchange rate is, but also how to exchange currency in large volumes.

That said, if it can solve these problems the fee that the service receives on every transaction makes it a potentially lucrative business because it solves the mismatch between customers' desire to pay bitcoins and merchants' desire to just get dollars and concentrate on selling goods.

4.6 Transaction Fees

The topic of transaction fees has come up in previous chapters and it will come up again in later chapters. Here we'll discuss the practical details of how transaction fees are set in Bitcoin today.

Whenever a transaction is put into the Bitcoin block chain, that transaction might include a transaction fee. Recall from a previous chapter that a transaction fee is just defined to be the difference between the total value of coins that go into a transaction minus the total value of coins that come out. The inputs always have to be at least as big as the outputs because a regular transaction can't create coins, but if the inputs are bigger than the outputs then the difference is deemed to be a transaction fee, and that fee goes to the miner who makes the block that includes this transaction.

The economics of transaction fees are interesting and we'll come back to this in a later chapter, but for now let’s see how transaction fees are actually set in Bitcoin as it operates as of early 2015. These details do change from time to time, but we'll give you a snapshot of the current state.

Why do transaction fees exist at all? The reason is that there is some cost that someone has to incur in order to relay your transaction. The Bitcoin nodes need to relay your transaction and ultimately a miner needs to build your transaction into a block, and it costs them a little bit to do that. For example, if a miner’s block is slightly larger because it contains your transaction, it will take slightly longer to propagate to the rest of the network and there’s a slightly higher chance that the block will be orphaned if another block was found near-simultaneously by another miner.

So, there is a cost — both to the peer to peer network and to the miners — of incorporating your transaction. The idea of a transaction fee is to compensate miners for those costs they incur to process your transaction. Nodes don’t receive monetary compensation in the current system, although running a node is of course far less expensive than being a miner. Generally you're free to set the transaction fee to whatever you want it to be. You can pay no fee, or if you like you can set the fee quite high. As a general matter, if you pay a higher transaction fee it's natural that your transaction will be relayed and recorded more quickly and more reliably.

Current default transaction fees. The current transaction fees that most miners expect is as follows: first of all, no fee is charged if a transaction meets all of these three conditions:

1. the transaction is less than 1000 bytes in size,
2. all outputs are 0.01 BTC or larger
3. priority is large enough

Priority is defined as: (sum of input age * input value) / (transaction size). In other words, look at all of the inputs to the transaction, and for each one compute the product of that input’s age and its value in bitcoins, and add up all those products. Note that the longer a transaction output sits unspent, the more it ages, and the more it will contribute to priority when it is finally spent.

If you meet these three requirements then your transaction will be relayed and it will be recorded in the block chain without a fee. Otherwise a fee is charged and that fee is about .0001 BTC per 1000 bytes, and as of this writing that's a fraction of a U.S. penny per 1000 bytes. The approximate size of a transaction is 148 bytes for each input plus, 34 bytes for each output plus, and ten bytes for other information. So a transaction with two inputs and two outputs would be about 400 bytes.

The current status quo is that most miners enforce the above fee structure, which means that they will either not service or will service last transactions that don't provide the necessary transaction fees. But there are other miners who don't enforce these rules, and who will record and operate on a transaction even if it pays a smaller fee or no fee at all.

If you make a transaction that doesn't meet the fee requirements it will probably find it's way into the block chain anyway, but the way to get your transaction recorded more quickly and more reliably is to pay the standard fee, and that's why most wallet software and most payment services include the standard fee structure in the payments that go on, and so you'll see a little bit of money raked off for transaction fees when you engage in everyday Bitcoin business.

4.7 Currency Exchange Markets

By currency exchange we mean trading bitcoins against fiat currency like dollars and euros. We've talked earlier about services that let you do this, but now we want to look at this as a market — its size, extent, how it operates, and a little bit about the economics of this market.

The first thing to understand is that it operates in many ways like the market between two fiat currencies such as dollars and euros. The price will fluctuate back and forth depending on how badly people want to buy euros versus how badly people want to buy dollars on a particular day. In the Bitcoin world there are sites like bitcoincharts.com that shows the exchange rate with various fiat currencies on a number of different exchanges.

As you'll see if you explore the site, there's a lot of trading going on, and the prices move in real time as trades are made. It's a liquid market and there are plenty of places that you can go to to buy or sell bitcoins. In March 2015 the volume on Bitfinex, the largest Bitcoin — USD exchange, was about 70,000 bitcoins or about 21 million dollars over a 24 hour period.

Another option is to meet people to trade bitcoins in real life. There are sites that help you do this. On localbitcoins.com, for example, you can specify your location and that you wish to buy bitcoins with cash. You'll get a bunch of results of people who at the time of your search are willing to sell bitcoins at that location, and in each case it tells you what price and how many bitcoins they're offering. You can then contact any of them and arrange to meet at a coffee shop or in a park or wherever, give them dollars and receive bitcoins in exchange. For small transactions, it may be sufficient to wait for one or two confirmations on the block chain.

Finally, in some places there are regular meet-ups where people go to trade bitcoins, and so you can go to a certain park or street corner or cafe at a scheduled day and time and there will be a bunch of people wanting to buy or sell bitcoins and you can do business with them. One reason someone might prefer obtaining bitcoins in person over doing so online is that it's anonymous, to the extent that a transaction in a public place can be considered anonymous. On the other hand, opening an account with an exchange generally requires providing government-issued ID due to banking regulation. We'll discuss this in more detail in Chapter 7.

Supply and demand. Like any market, the Bitcoin exchange market matches buyers who want to do one thing with sellers that are willing to do the opposite thing. It's a relatively large market — millions of U.S. dollars per day pass through it. It's not at the scale of the New York Stock Exchange or the dollar–euro market, which are vastly larger, but it's large enough that there is a notion of a consensus price. A person who wants to come into this market can buy or sell at least a modest amount and will always be able to find a counterparty.

The price of this market, this consensus price, like the price of anything in a liquid market will be set by supply and demand. By that we mean the supply of bitcoins that might potentially be sold and the demand for bitcoins by people who have dollars. The price through this market mechanism will be set to the level that matches supply and demand. Let's dig into this in a little more detail.

What is the supply of bitcoins? This is the number of bitcoins that you might possibly buy in one of these markets, and it is equal to the supply of bitcoins that are in circulation currently. There's a fixed number of bitcoins in circulation. At the time of this writing it's about 13.9 million, and the rules of Bitcoin as they currently stand say that this number will slowly go up and eventually hit a limit of 21 million.

You might also sometimes include demand deposits of bitcoins. That is, if someone has put money into their account in a Bitcoin exchange, and the exchange doesn't keep a full reserve to meet every single deposit, then you'll have demand deposits at that exchange that are larger than the number of coins that the exchange is holding, and depending on what question you're asking about the market it might or might not be correct to include demand deposits in the supply.

So, when should you include demand deposits? Well, basically, you should include demand deposits in a market analysis when demand-deposited money can be sold in that market. For example, if you're talking about exchange of dollars for bitcoins that can happen in an exchange, and the exchange allows demand-deposited bitcoins to be traded for dollars, then they count.

It's worth noting, as well, that when economists conventionally talk about the supply of fiat currency they typically include in the money supply not only the currency that's in circulation — that is, paper and metal money — but also the total amount of demand deposits, and that's for the logical reason that people can actually spend their demand-deposited money to buy stuff. So although it's tempting to say that the supply of bitcoins is fixed at 13.1 million currently or 21 million eventually, for some purposes we have to include demand deposits where those demand deposits function like money, and so the supply might not be fixed the way some Bitcoin advocates might claim. We need to look at the circumstances of the particular market we're talking about in order to understand what the proper money supply is. But let's assume we've agreed on what supply we're using based on what market we're analyzing.

Let's now look at demand. There are really two main sources of demand for bitcoins. There's a demand for bitcoins as way of mediating fiat currency transactions and there's demand for bitcoins as an investment.

First let's look at mediating fiat currency transactions. Imagine that Alice wants to buy something from Bob and wants to pay some money to Bob, and Alice and Bob want to transfer let's a say a certain amount of dollars, but they find it convenient to use Bitcoin to do this transfer. Let’s assume here that neither Alice nor Bob is interested in holding bitcoins long-term. We’ll return to that possibility in a moment. So Alice would buy bitcoins for dollars and transfer them, and once they receive enough confirmations to Bob's satisfaction, he’ll sell those bitcoins for dollars. The key thing here from the point of view of demand for bitcoins is that the bitcoins mediating this transaction have to be taken out of circulation during the time that the transaction is going on. This creates a demand for bitcoins.

The second source of demand is that Bitcoin is sometimes demanded as an investment. That is if somebody wants to buy bitcoins and hold them in the hope that the price of bitcoins will go up in the future and that they'll be able to sell them. When people buy and hold, those bitcoins are out of circulation. When the price of Bitcoin is low, you might expect a lot of people to want to buy bitcoins as an investment, but if the price goes up very high then the demand for bitcoins as an investment won't be as high.

A simple model of market behavior. Now, we can do some simple economic modeling to understand how these markets will behave. We won’t do a full model here although that's an interesting exercise. Let’s look specifically at the the transaction-mediation demand and what effect that might have on the price of bitcoins.

We’ll start by assuming some parameters. T is the total transaction value mediated via Bitcoin by everyone participating in the market. This value is measured in dollars per second. That’s because we assume for simplicity that the people who want to mediate these transactions have in mind a certain dollar value of the transactions, or some other fiat currency that we'll translate into dollars. So there's a certain amount of dollars per second of transactions that need to be mediated. D is the duration of time that bitcoins need to be held out of circulation in order to mediate a transaction. That's the time from when the payer buys the bitcoins to when the receiver is able to sell them back into the market, and we'll measure that in seconds. S is the total supply of bitcoins that are available for this purchase, and so that's going to be all of the hard-currency bitcoins that exist — currently about 14 million or eventually up to 21 million — minus those that are held out by people as long term investments. In other words, we’re talking about the bitcoins sloshing around and available for mediating transactions purpose. Finally, P is the price of Bitcoin, measured in dollars per bitcoin.

Now we can do some calculations. First we'll calculate how many bitcoins become available in order to service transactions every second. There are S bitcoins available in total and because they're taken out of circulation for a time of D seconds, every second on average an S/D fraction of those bitcoins will become newly available because they'll emerge from the out-of-circulation state and become available for mediating transactions every second. That's the supply side.

On the demand side — the number of bitcoins per second that are needed to mediate transactions — we have T dollars worth of transactions to mediate and in order to mediate one dollar worth of transactions we need 1/P bitcoins. So T/P is number of bitcoins per second that are needed in order to serve all of the transactions that people want to serve.

Now if you look at a particular second of time, for that second there's a supply of S/D and a demand of T/P. In this market, like most markets, the price will fluctuate in order to bring supply into line with demand. If the supply is higher than the demand then there are bitcoins going unsold, so people selling bitcoins will be willing to lower their asking price in order to sell them. And according to our formula T/P for demand, when the price drops the demand increases, and supply and demand will reach equilibrium.

On the other hand, if supply is smaller than demand it means that there are people who want to get bitcoins in order to mediate a transaction but can't get them because there aren't enough bitcoins around. Those people will then have to bid more in order to get their bitcoins because there will be a lot of competition for a limited supply of bitcoins. This drives the price up, and referring to our formula again, it means that demand will come down until there is equilibrium. In equilibrium, the supply must equal the demand, so we have

$$\frac{S}{D} = \frac{T}{P}$$

which gives us a formula for the price:

$$P = \frac{TD}{S}$$

What does this equation tell us? We can simplify it a bit further: we can assume that D, the duration for which you need to hold a bitcoin to mediate a transaction, doesn’t change. The total supply S also doesn’t change, or at least changes slowly over time. That means the price is proportional to the demand for mediation as measured in dollars. So if the demand for mediation in dollars doubles then the price of bitcoins should double. We could in fact graph the price against some estimate of the demand for transaction mediation and see whether or not they match up. When economists do this, the two do tend to match up pretty well.

Note is that the total supply S includes only the bitcoins that aren't being held as investments. So if more people are buying bitcoins as an investment, S will go down, and our formula tells us that P will go up. This makes sense — if there's more demand on the investment side then the price that you need to pay to mediate a transaction will go up.

Now this is not a full model of the market. To have a full model we need to take into account the activity of investors. That is, investors will demand bitcoins when they believe the price will be higher in the future, and so we need to think about investors’ expectations. These expectations, of course, have something to do with the expected demand in the future. We could build a model that is more complex and takes that into account, but we won’t do that here.

The bottom line here is that there is a market between bitcoins and dollars, and between bitcoins and other fiat currencies. That market has enough liquidity that you can buy or sell in modest quantities in a reliable way, although the price does go up and down. Finally, it's possible to do economic modeling and have some idea about how supply and demand interact in this market and predict what the market might do, as long as you have a way to estimate unknowable things like how much are people going to want to use Bitcoin to mediate transactions in the future. That kind of economic modeling is important to do and very informative, and surely there are people who are doing it in some detail today, but a detailed economic model of this market is beyond the scope of this text.

Further reading

Securing bitcoins has some similarities, as well as important differences, to the way banks secure money. Chapter 10 of Ross Anderson's security textbook, titled "Banking and bookkeeping", is a great read. The entire book is freely available online.

Anderson, Ross. *Security engineering*. John Wiley & Sons, 2008.

The study analyzing closures of Bitcoin exchanges that we referenced:

Moore, Tyler, and Nicolas Christin. *Beware the middleman: Empirical analysis of bitcoin-exchange risk.* Financial Cryptography and Data Security 2013.

Adi Shamir's paper on secret sharing:

Shamir, Adi. *How to share a secret*. Communications of the ACM 22.11 (1979).

Exercises

1. **Proof of reserve.** TransparentExchange claims that it controls at least 500,000 BTC and wants to prove this to its customers. To do this it publishes a list of addresses that have a total balance of 500,000 BTC. It then signs the statement "TransparentExchange controls at least 500,000 BTC" with each of the corresponding private keys, and presents these signatures as proof.
   What are some ways in which TransparentExchange might be able to produce such a proof even if it doesn't actually currently control 500,000 BTC? How would you modify the proof to make it harder for the exchange to cheat?
2. **Proof of liabilities.**
   TransparentExchange implements a Merkle Tree based protocol to prove an upper bound on its total deposits. (Combined with a proof of reserve, this proves that the exchange is solvent.) Every customer is assigned a leaf node containing an ID which is the hash of her username and a value which is her BTC balance. The protocol specifies that TransparentExchange should propagate IDs and values up the tree by the following recursive definition — for any internal node:

```
node.value = node.left_child.value + node.right_child.value
```

node.id = Hash(node.left_child.id || node.right_child.id || node.value)

The exchange publishes the root ID and value, and promises to prove to any customer that her node is included in the tree (by the standard Merkle tree proof of inclusion). The idea is that if the exchange tries to claim a lower total than the actual sum of deposits by leaving some customers out of the tree or by making their node value less than their balance, it will get caught when any of those customers demand a proof of inclusion.

- **2.1.** Why can't the exchange include fake customers with negative values to lower the total?
- **2.2.** Show an attack on this scheme that would allow the exchange to claim a total less than the actual sum of deposits.
- **2.3.** Fix this scheme so that is not vulnerable to the attack you identified.
- **2.4.** Ideally, the proof that the exchange provides to a customer shouldn't leak information about other customers. Does this scheme have this property? If not, how can you fix it?

3. Transaction fees.

- **3.1.** Alice has a large number of coins each of small value *v*, which she would like to combine into one coin. She constructs a transaction to do this, but finds that the transaction fee she'd have to spend equals the sum of her coin values. Based on this information (and the default transaction fee policy specified in slide 50), estimate *v*.
- **3.2.** Can Alice somehow consolidate her coins without incurring any transaction fee under the default policy?
- **3.3.** Compared to a fee structure that doesn't factor the age of the inputs into the transaction fee, what effect might the current default fee structure have on the behavior of users and services?

4. Multi-signature wallet

- **4.1.** BitCorp has just noticed that Mallory has compromised one of their servers holding their Bitcoin private keys. Luckily, they are using a 2-of-3 multi-signature wallet, so Mallory has learnt only one of the three sets of keys. The other two sets of keys are on different servers that Mallory cannot access. How do they re-secure their wallet and effectively revoke the information that Mallory has learned?
- **4.2.** If BitCorp uses a 2-out-of-2 instead of a 2-out-3 wallet, what steps can they take in advance so that they can recover even in the event of one of their servers getting broken into (and Mallory not just learning but also potentially deleting the key material on that server)?

5. Exchange rate

- **5.1.** Speculate about why buying bitcoins in person generally more expensive than buying from an online exchange.
- **5.2.** Moore and Christin observe that security breaches and other failures of exchanges have little impact on the Bitcoin exchange rate. Speculate on why this might be.

6. Payments. A Bitcoin payment service might receive thousands of payments from various users near-simultaneously. How can it tell whether a particular user Alice who logged into the payment service website and initiated the payment protocol actually made a payment or not?

7. **BitcoinLotto:** Suppose the nation of Bitcoinia has decided to convert its national lottery to use Bitcoin. A trusted scratch-off ticket printing factory exists and will not keep records of any values printed. Bitcoinia proposes a simple design: a weekly run of tickets is printed with an address holding the jackpot on each ticket. This allows everybody to verify the jackpot exists. The winning ticket contains the correct private key under the scratch material.
   - **7.1.** What might happen if the winner finds the ticket on Monday and immediately claims the jackpot? Can you modify your design to ensure this won't be an issue?
   - **7.2.** Some tickets inevitably get lost or destroyed. So you'd like to modify the design to roll forward any unclaimed jackpot from Week *n* to the winner in Week *n+1*. Can you propose a design that works, without letting the lottery administrators embezzle funds? Also make sure that the Week *n* winner can't simply wait until the beginning of Week *n+1* to attempt to double their winnings.

Bitcoin and Cryptocurrency Technologies

Arvind Narayanan, Joseph Bonneau, Edward Felten,
Andrew Miller, Steven Goldfeder

Draft — Apr 10, 2015

Feedback welcome! Email bitcoinbook@lists.cs.princeton.edu

Chapter 5: Bitcoin Mining

This chapter is all about mining. We've already seen quite a bit about miners and how Bitcoin relies on them — they validate every transaction, they build and store all the blocks, and they reach a consensus on which blocks to include in the block chain. We also have already seen that miners earn some reward for doing this, but we still have left many questions unanswered. Who are the miners? How did they get into this? How do they operate? What's the business model like for miners? What impact do they have on the environment? In this chapter, we will answer all of these questions.

5.1 The task of Bitcoin miners

Do you want to get into Bitcoin mining? If you do, we're not going to completely discourage you, but beware that Bitcoin mining bears many similarities to gold rushes. Historical gold rushes are full of stories of young people rushing off to find fortune, and many of them lose everything they have. A few strike it rich, but even those that do generally endure lots of hardship along the way. Flocking to a gold rush isn't easiest way to get rich, and Bitcoin mining is starting to look like a similar proposition. As we'll see in this section, mining is by no means a get-rich-quick scheme.

But first, let's look at the technical details. To be a Bitcoin miner, you have to join the Bitcoin network and connect to other nodes. Once you're connected, there are six tasks to perform:

1. *Listen for transactions.* First, you listen for transactions on the network and validate them by checking the signatures and that the outputs being spent haven't been spent before.
2. *Maintain block chain and listen for new blocks.* You must maintain the block chain. You start by requesting other nodes to give you all of the historical blocks that are already part of the block chain before you joined the network. You then listen for new blocks that are being broadcast to the network. You must validate each block that you receive — by validating each transaction in the block and checking that the block contains a valid nonce. We'll return to the details of nonce checking later in this section.
3. *Assemble a new block.* Once you have an up-to-date copy of the block chain, you begin building your own blocks. To do this, you group transactions that you heard about into a new block that extends the latest block you know about. You must make sure that each transaction included in your block is valid.
4. *Find a nonce that makes your block valid.* This step requires the most work, and it's where all the difficulty really happens for the miners. We will see this in detail shortly.
5. *Hope your block is accepted.* Even if you found a block, there's no guarantee that your block will become part of the consensus chain. There's bit of luck here; you have to hope that other miners accept your block and start mining on top of it, instead of some competitor's block.
6. *Profit.* If all other miners do accept your block, then you profit! At the time of this writing in early 2015, the block reward is 25 bitcoins which is currently worth over $6,000. In addition, if any of the transactions in the block contained transaction fees, the miner collects those too.

We can classify the steps that a miner must take into two categories. Some tasks — validating transactions and blocks — help the Bitcoin network and are fundamental to its existence. These tasks are the reason that the Bitcoin protocol requires miners in the first place. Other tasks — the race to find blocks and profit —- aren't necessary for the Bitcoin network itself but are intended to incentivize miners to perform the essential steps. Of course, both of these are necessary for Bitcoin to function as a currency, since miners need an incentive to perform the critical steps.

Finding a valid block. Let's return to the question of finding a nonce that makes your block valid. In Chapter 3 we saw that there are two main hash-based structures. There's the block chain where each block header points to the previous block header in the chain, and then within each block there's a Merkle tree of all of the transactions included in that block.

The first thing that you do as a miner is you assemble all the transactions that you have from your pending transaction pool into a Merkle tree. You then create a block with a header that points to the previous block. In the block header, there's a 32 bit nonce field, and you keep trying different nonces looking for one that that causes the block's hash to be under the target — roughly, begin with the required number of zeros. A miner may begin with a nonce of 0 and successively increment it by one in search of a nonce that makes the block valid. See Figure 5.1.

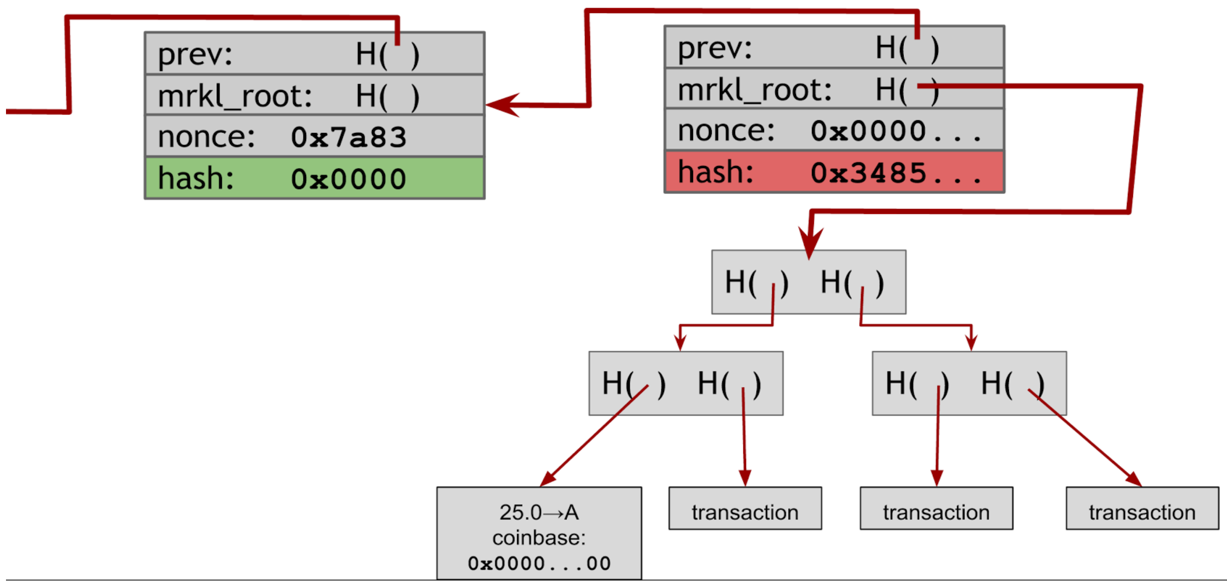


Figure 5.1: Finding a valid block. In this example, the miner tries a nonce of all 0s. It does not produce a valid hash output, so the miner would then proceed to try a different nonce.

In most cases you'll try every single possible 32-bit value for the nonce and none of them will produce a valid hash. At this point you're going to have to make further changes. Notice in Figure 5.1 that there's an additional nonce in the coinbase transaction that you can change as well. After you've exhausted all possible nonces for the block header, you'll change the extra nonce in the coinbase transaction — say by incrementing it by one — and then you'll start searching nonces in the block header once again.

When you change the nonce parameter in the coinbase transaction, the entire Merkle tree of transactions has to change (See Figure 5.2). So the change of the coinbase nonce will propagate all the way up, and since you'll have to update all the hashes, changing the extra nonce in the coinbase transaction is much more expensive than changing the nonce in the block header. For this reason, miners spend most of the time changing the nonce in the block header and only change the coinbase nonce when they have exhausted all of the 2^{32} nonces in the block header.

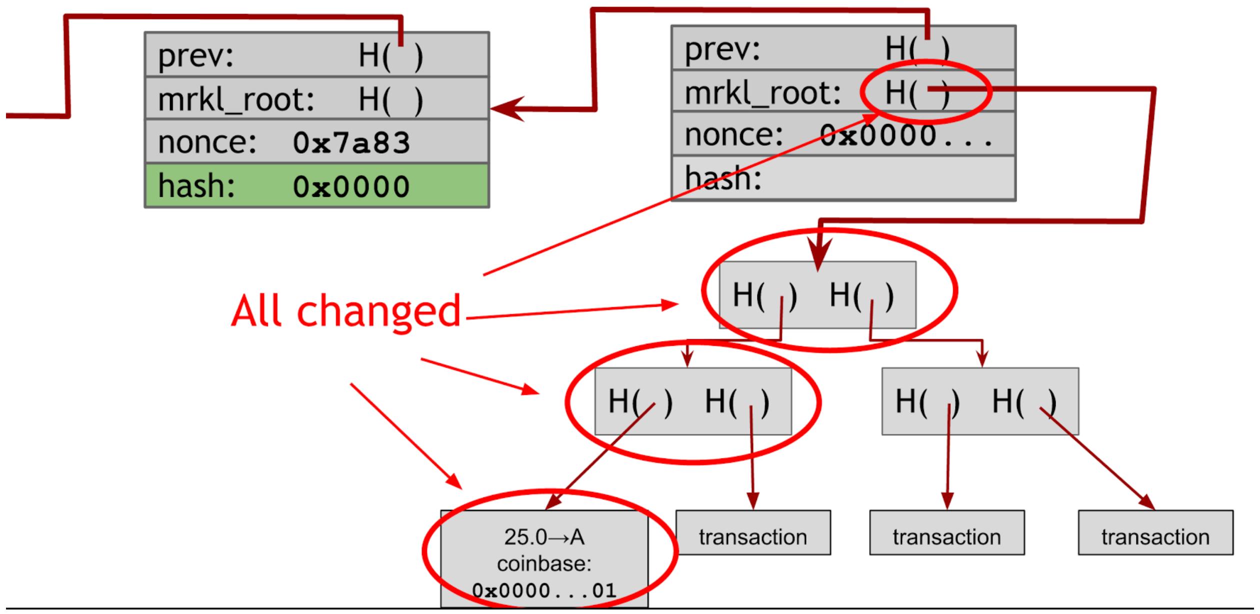


Figure 5.2: Changing a nonce in the coinbase transaction propagates all the way up the Merkle tree.

The vast, vast majority of nonces that you try aren't going to work, but if you stay at it long enough you'll eventually find the right combination of the extra nonce in the coinbase transaction and the nonce in the block header that produce a block with a hash under the target. When you find this, you want to announce it as quickly as you can and hope that you can profit from it.

Is everyone solving the same puzzle? You may be wondering: if every miner just increments the nonces as we described, aren't all miners solving the exact same puzzle? Won't the fastest miner always win? The answer is no! Firstly, it's unlikely that miners will be working on the exact same block as each miner will likely include a somewhat different set of transactions and in a different order. But more importantly, even if two different miners were working on a block with identical transactions, the blocks would still differ. Recall that in the coinbase transaction, miners specify their own address. This change will propagate up causing all the Merkle hashes to change ensuring that no two miners are hashing the same inputs.

Difficulty. Exactly how difficult is it to find a valid block? As of March 2015, the mining difficulty target (in hexadecimal) is:

0000000000000000172EC00000000000000000000000000000000000000000000000

so the hash of any valid block has to be below this value. In other words only one in about 2^{67} nonces that you try will work, and that's a really huge number. One approximation for it that you would think about is it's about the population of the earth squared. So, if every person on Earth was themselves their own planet Earth with seven billion people on it the total number of people would be close to this number.

Determining the difficulty. The mining difficulty changes every 2016 blocks. It is adjusted based on how efficient the miners were over the period of the previous 2016 blocks according to this formula:

next_difficulty = previous_difficulty * (2 weeks) / (time to mine last 2016 blocks)

Two weeks is the amount of time it would take to mine 2016 if a block were created exactly every 10 minutes. So the effect of this formula is to scale the difficulty to maintain the property that blocks should be found by the network on average about once every ten minutes. There's nothing special about 2 weeks, but it's a good trade-off. If the period were much shorter, the difficulty might fluctuate due to random variations in the number of blocks found in each period. If the period were much higher, the network's hash power might get too far out of balance with the difficulty.

Each Bitcoin miner independently computes the difficulty and will only accept blocks that meet the difficulty that they computed. Miners who are on different branches might not compute the same difficulty value, but any two miners mining on top of the same block will agree on what the difficulty should be. This allows consensus to be reached.

You can see in Figure 5.3 that over time the mining difficulty keeps increasing. It's not necessarily a steady linear increase or an exponential increase, but it depends on activity in the market. Things like how many new miners are joining, which in turn may be affected by the current exchange rate of Bitcoin, affect the mining difficulty. Generally, as more and more miners come online, blocks are found faster, and the difficulty is increased so that it again takes ten minutes to find a block.

In Figure 5.3 you can see that in the red line on the graph there's a step function of difficulty even though the overall network hash is growing smoothly. The discrete step results from the fact that the difficulty is only adjusted every 2016 blocks.

Another way to view this is to view how long it takes to find a block on average. Figure 5.4 shows how many seconds elapse between consecutive blocks in the block chain. You can see that this gradually goes down, jumps up and then gradually goes down again. Of course what's happening is that every

2016 blocks the difficulty resets and the average block time goes back up to about ten minutes. Over the next period the difficulty stays unchanged, but more and more miners come online. Since the hash power has increased but the difficulty has not, blocks are found more quickly until the difficulty is again adjusted after 2016 blocks, or about two weeks.

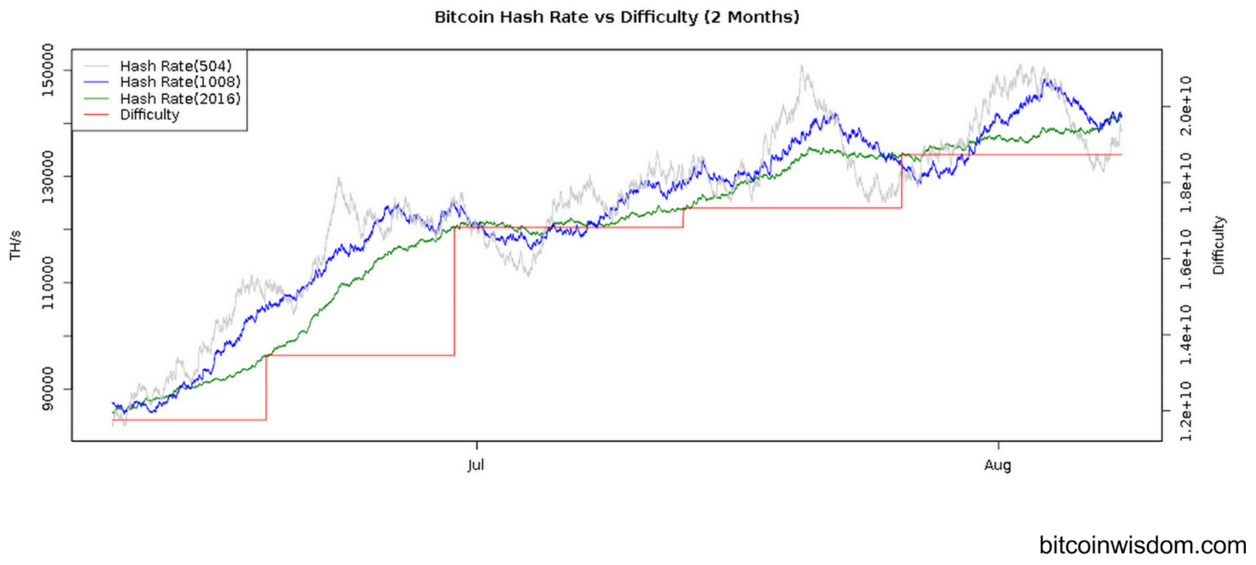


Figure 5.3: Mining difficulty over time (mid-2014). Note that the y-axis begins at 80,000 TH/s.

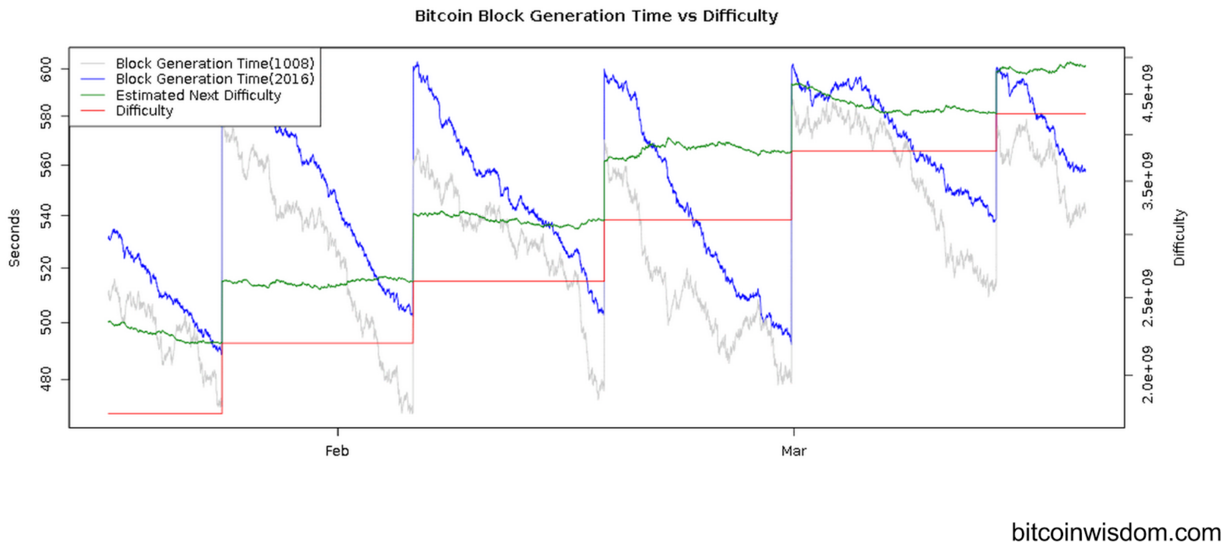


Figure 5.4 : Time to find a block (early 2014). Note that the y-axis begins at 460 seconds.

Even though the goal was for a block to be found every ten minutes on average, it's actually close to about every nine minutes, and at the end of the two week cycle it will get down to around eight minutes. This behavior is during a period of rapid hash rate increase. If the hash rate isn't increasing as fast, the average time to find a block will be more stable.

There have been *decreases* in difficulty a few times in Bitcoin's history, small in magnitude compared to its increases. One proposed scenario for Bitcoin's collapse is a "death spiral" in which a dropping exchange rate makes mining unprofitable for some miners, causing an exodus, in turn causing the price to drop further. While there have been no catastrophic declines of mining power so far, there's no inherent reason why difficulty must keep increasing.

5.2 Mining Hardware

We've mentioned that the computation that miners have to do is very difficult. In this section, we'll discuss why it is so computationally difficult and take a look at the hardware that miners use to facilitate this computation.

What exactly is this difficult computation that miners are working on? They are computing many, many SHA-256 hashes. We've discussed hash functions and we've mentioned SHA-256 in particular. SHA-256 is a general purpose cryptographic hash function that's part of a bigger family of functions that was standardized in 2001. SHA-256 was a good choice as this was strongest cryptographic hash function available at the time when Bitcoin was designed. It is possible that it will become less secure over the lifetime of Bitcoin, but for now it remains secure. It did come out of the NSA, which has led to some conspiracy theories, but it's generally considered to be a very strong hash function.

Sidebar. Although SHA-256 is generally considered to be cryptographically secure, its replacement, the SHA-3 family, has already been picked. SHA-3 is in the final stages of standardization today, but it wasn't available at the time Bitcoin was designed.

A closer look at SHA-256. Figure 5.5 shows more detail about what actually goes on in a SHA-256 computation. While we don't need to know all of the details to understand how Bitcoin works, we'll give a high level overview so you have a general idea of the task that miners are solving.

SHA-256 maintains 256 bits of state. The state is split into eight 32-bit words which makes it very optimized for 32-bit hardware, and in each round some bitwise tweaks that are applied to some of those words. Then a number of words in the state are taken — some with these tweaks applied — and added together mod 32. The result of all of these additions is wired over to the first word of the state and the entire state shifts over.

Figure 5.5 is just one round of the SHA-256 compression function, and a complete computation of SHA-256 does this for 80 iterations. During each round, there are slightly different constants applied so that every iteration isn't exactly the same.

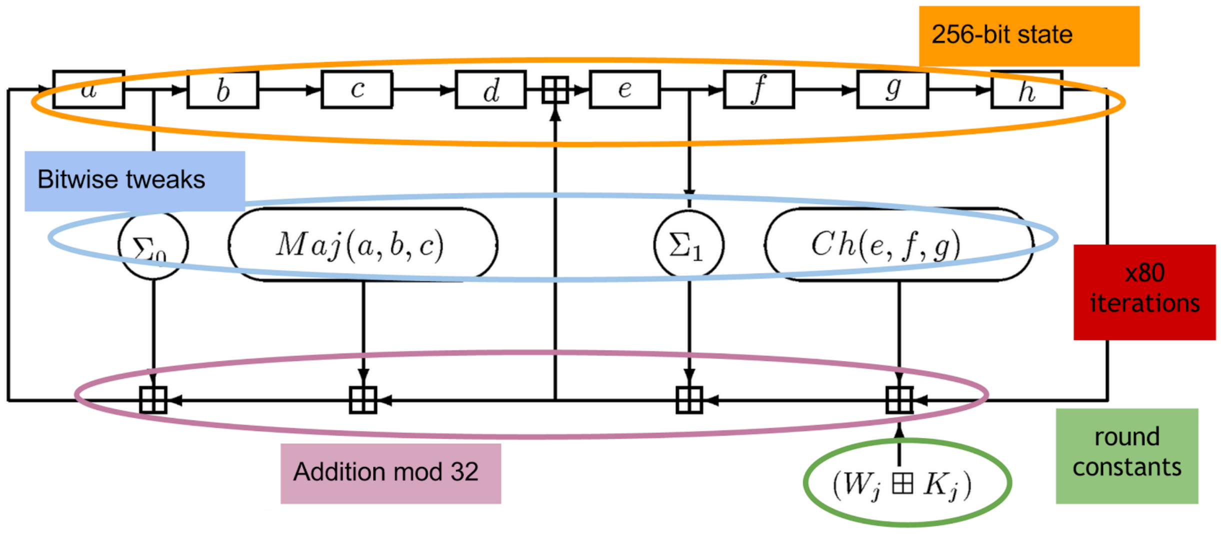


Figure 5.5 : The structure of SHA-256. This is one round of the compression function.

So the task for miners is compute this function. To do this, they need to be able to deal with 32-bit words, do 32-bit modular addition, and also be able to do some bitwise logic. Remember that miners are racing each other so they will want to do this as fast as possible.

As we will see shortly, Bitcoin actually requires SHA-256 to be applied twice to a block in order to get the hash that is used by the nodes. This is a quirk of Bitcoin, and the reason for the double application are not fully specified and seemingly unnecessary, but at this point, it's just something that miners have to deal with.

CPU mining. The first generation of mining was all done on general purpose computers — that is general purpose central processing units (CPUs). In fact, CPU mining was as simple as running the code shown in Figure 5.6. That is, miners simply searched over nonces in a linear fashion, computed SHA 256 in software and checked if the result was a valid block. Also, notice in the code that as we mentioned, SHA-256 is applied twice.

```
while (1) {
      HDR[kNoncePos]++;
      IF (SHA256(SHA256(HDR)) < (65535 << 208) / DIFFICULTY)
            return;
}
```

Figure 5.6 : CPU mining pseudocode.

How fast will this run on a general purpose computer? On a high end desktop PC you can compute about 20 million hashes per second (MH/s). At that speed, it would take you over a hundred thousand years on average at the early-2015 difficulty level to find a block. We talked about how mining was going to be a difficult slog, and if you're mining on a general purpose PC today it's a really, really big hill to get up because it's going to take you about 300,000 years on average to find a block. CPU mining is no longer profitable with the current difficulty. For the last few years, anyone trying mine on a CPU probably doesn't understand how Bitcoin works and were probably pretty disappointed that they never made any money doing it.

GPU mining. The second generation began when people started to get frustrated with how slow their CPUs were and instead used their graphics card, or graphics processing unit (GPU).

Almost every modern computer has a GPU built in for high performance graphics. They're designed to have high throughput, and also high parallelism, both of which are very useful for Bitcoin mining. Bitcoin mining can be parallelized because you can compute multiple hashes at the same time with different nonces. In 2010, a language called OpenCL was released. OpenCL is a general purpose language to do things other than graphics on a GPU. It's a high level-language, but over time people started tweaking the code even further to run more quickly on specific graphics cards. This paved the way for Bitcoin mining on GPUs.

Mining with graphics cards has some nice properties. For one thing, they're easily available, and they're easy for amateurs to set up. You can order graphics cards online or buy them at most big consumer electronics stores. They're the most accessible high-end hardware that's available to most people. They also have some properties that make them specifically good for Bitcoin mining. They're designed for parallelism so they have a lot of Arithmetic Logic Units (ALUs) in them that you can use in parallel to do different SHA-256 computations, and some of them also have specific instructions to do bitwise operations that work out quite nicely for SHA-256. They also have the property that you can drive many graphics cards from one motherboard and CPU. So you could take your one computer and attach multiple graphics cards to it.

Most graphics cards can also be ***overclocked*** which is a property that gamers demand so you can run them faster than they're actually designed for, if you want to take on the risk. And with Bitcoin mining, it might be a good idea to run the chip much faster than it was designed for even if you introduce some errors in the process. For example, say you can run your graphics card 50 percent faster but doing so will increase the error in the SHA-256 computation to 30 percent of the time. If an invalid solution is erroneously declared valid by the graphics card — something that would happen rarely — you can always double-check it on your CPU. On the other hand, if a valid solution is erroneously missed, you'd never know. But if your speed increase from overclocking can overcome the decrease in output due to errors, you'd still come out ahead. There's a term called ***goodput*** that measures this, which is simply the product of throughput and success rate. In the above example, the throughput is 1.5x compared to not overclocking, whereas the success rate is 0.7x. The product is 1.05, which means overclocking increases the goodput by 5%. People have spent a long time optimizing exactly how much they should overclock a given chip and what errors it would introduce.

As we said earlier, you can control multiple GPUs from a single CPU, and people began taking advantage of this.They would use multiple graphics cards together for mining, and you began to see some really interesting home-brewed setups like this one shown in Figure 5.7. This was still in the early days of Bitcoin when miners were still mostly hobbyists who didn't know a lot about running a modern data center, but they came up with some quite ingenious designs for how to pack many graphics cards into a small place and keep them cool.

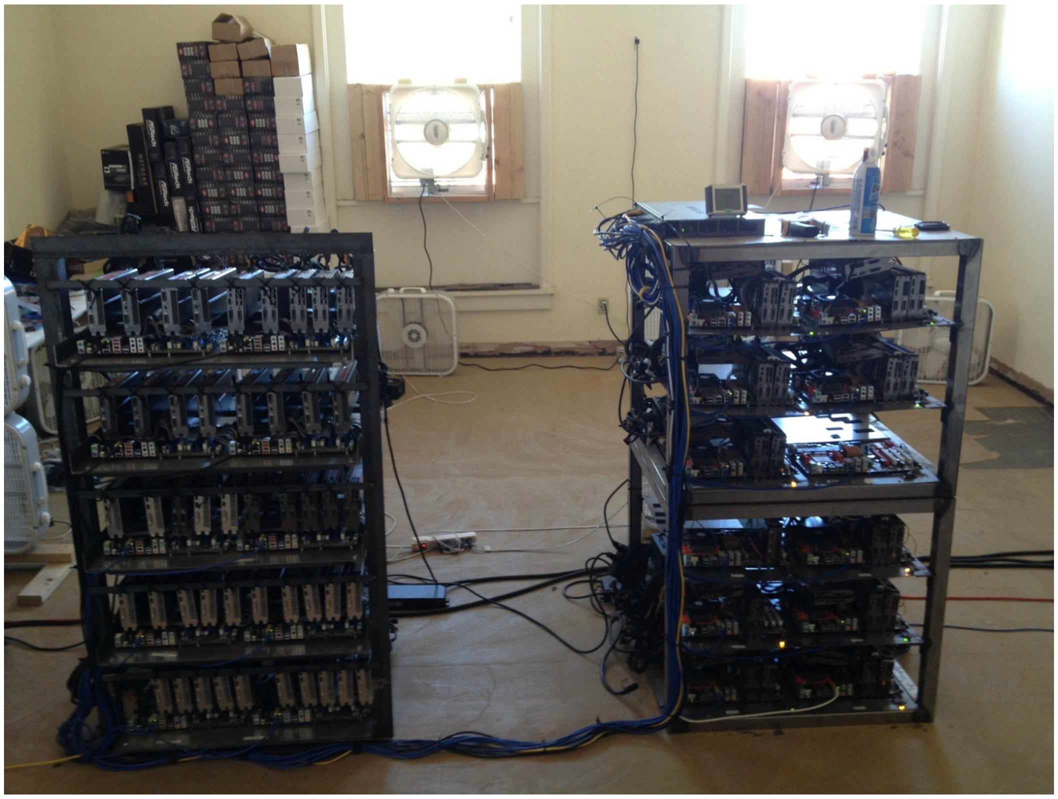


Figure 5.7: A home-built rack of GPUs used for Bitcoin mining. You can also see the fans that they used to build their cooling system. Source: LeonardH, cryptocurrenciestalk.com.

Disadvantages of GPU mining. GPU mining has some disadvantages. GPUs have a lot of hardware built into them for doing video that doesn't get used by miners. Specifically, they have floating point units that aren't used at all in SHA-256, and these are wasted when using them for mining. GPUs also don't have the greatest cooling characteristics when you put a lot of them next to one another. They're not designed to run side by side as they are in the picture; they're designed to be on one graphics card box doing graphics for one computer.

GPUs can also have a fairly large power draw, so a lot of electricity is being used relative to a computer. Another disadvantage initially was that you had to either build your own board or buy expensive boards to house multiple graphics cards.

On a really high-end graphics card with aggressive tuning you might get as high as 200 MH/s , or 200 million hashes per second, an order of magnitude better than you would be doing with a CPU. But even with that improved performance, and even if you're really aggressive and used one hundred

GPUs together, it would still take you over 300 years on average to find a block at the early-2015 difficulty level. Due to this lack of performance, GPU mining is basically dead.

FPGA mining. Around 2011 some miners started to use FPGAs or Field Programmable Gate Arrays. That's around the same time that the first implementation of Bitcoin mining came out in Verilog, a hardware design language that's used to program FPGAs. The rationale behind FPGAs is to try to get close to the performance characteristics of custom hardware while also allowing the owner of the card to customize it or reconfigure it "in the field." This lies in contrast to a chip which is made in a factory and does the same thing forever.

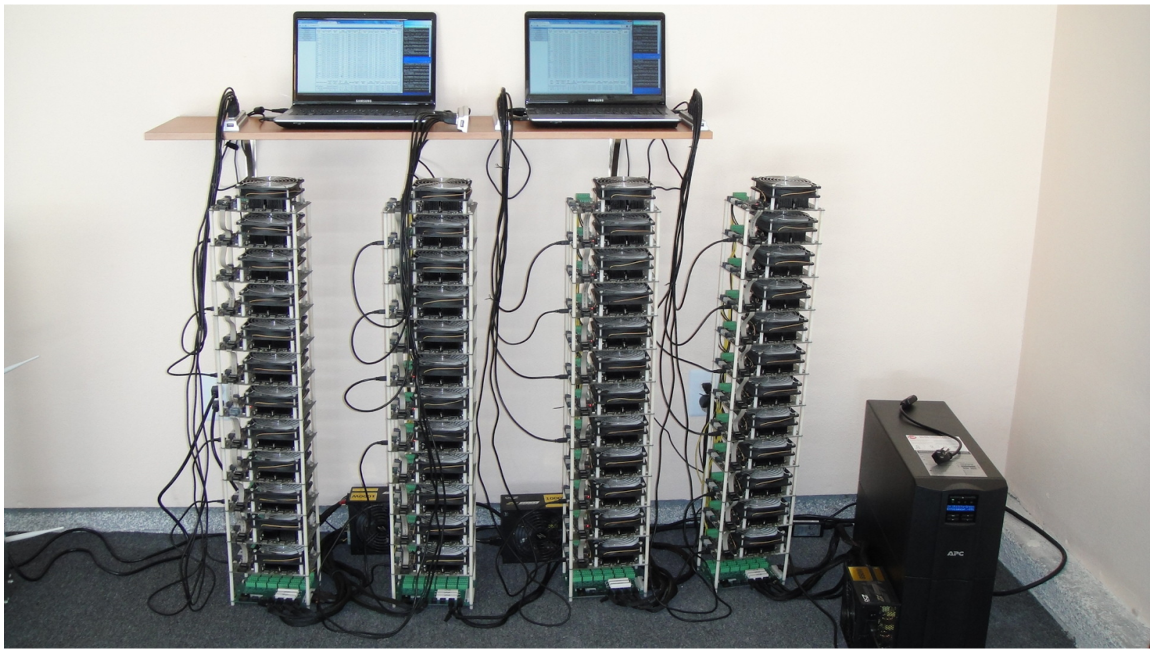


Figure 5.8: A home-built rack of FPGAs. Although you don't see the cooling setup pictured here, a rack like this would need a cooling system.

FPGAs offer better performance than graphics cards, particularly on some of the "bit fiddling" operations. These are easy to specify on an FPGA, and cooling is also easier with FPGAs. You're also wasting less of the card than you would be a graphics card. As with GPUs, you can pack many of these together and drive them from one central unit, and this is exactly what people began to do (see Figure 5.8). Overall, it was possible build a big array of FPGAs more neatly and cleanly than you could with graphics cards.

If you were using an FPGA and using it well, you might get up to a GH/s, or one billion hashes per second. This is certainly a large performance gain over CPUs and GPUs, but even if you had a hundred boards together, each with a 1 GH/s throughput, it would still take you about 50 years on average to find a Bitcoin block at the early-2015 difficulty level.

Despite the performance gain, the days of FPGA mining were quite limited. Firstly, they were being driven harder for Bitcoin mining — by being on all the time and overclocked — than a lot of consumer grade FPGAs were really designed for. Because of this, people found errors and malfunctions in their FPGAs as they were mining. It also turned out to be difficult to optimize the 32 bit addition step which is critical in doing SHA-256. FPGAs are also less accessible to people. You can't buy an FPGA at most stores, and there are few people who know how to program an FPGA or how to set them up.

Even though FPGAs improved performance, the cost-per-performance was only marginally improved over GPUs. FPGA mining was a rather short-lived phenomenon. Whereas GPU mining dominated for about a year or so, the days of FPGA mining were far more limited — lasting only a few months.

At this point you might be wondering that if all of these solutions are so intractable today, what are people actually using? This brings us to ASIC mining.

ASIC mining. Mining today is dominated by Bitcoin *ASICs*, or ***application-specific integrated circuits***. These are chips that were designed, built, and optimized for the sole purpose of mining Bitcoins. There are a few big vendors that that sell these to consumers. There is a good deal of variety in the ASICs that you can buy. You can choose between slightly bigger and more expensive models, more compact models, as well as models with varying performance and energy consumption claims.

Designing ASICs requires a lot of expertise and their lead-time is also quite long. Nevertheless, Bitcoin ASICs were designed and produced surprisingly quickly. In fact, analysts have said that this may be the fastest turnaround time in the history of integrated circuits for specifying a problem and turning it around to have a working chip in people's hands. On the flip side, the first few generations of Bitcoin ASICs were quite buggy, and most of them didn't quite deliver the promised performance numbers. Bitcoin ASICs have since matured, and there are now fairly reliable ASICs being shipped.

Up until 2014, the lifetime of ASICs has been quite short due to the rapidly increasing network hash rate, and thus shipping speed is crucial. Most boards in the ASIC era have been effectively obsolete in about six months. Furthermore, the bulk of the profits are made up front. Often, miners will make half of the expected profits for the lifetime of the ASIC during just the first six weeks. Due to the immaturity of the industry, consumers have often experienced shipping delays, with boards often obsolete by the time they reach the customer. If and when the growth rate of Bitcoin’s hash power stabilizes, mining equipment will have a longer life time.

For much of Bitcoin’s history, the economics of mining haven’t been favorable to the small miner who wants to go online, order mining equipment, and start making money. In fact, in most cases people who have placed orders for mining hardware should have lost money based on the calculation that they made at the time. Until 2013, the price of Bitcoin rose a lot, and this saved most of those customers from losing money. In effect, mining has been an expensive way to simply bet that the price of Bitcoin would rise, and a lot of miners — even though they've made money mining Bitcoins — would have been better off if they had just taken the money that they were going to spend on mining equipment, invested it in Bitcoin, and eventually sold them at a profit.

Today : Professional mining. Today mining has mostly moved away from individuals and toward professional mining centers. Exact details about how these centers operate are not very well known because companies want to protect their setups to maintain a competitive advantage. In Figure 5.9, we see a picture of a professional mining center in the Republic of Georgia.

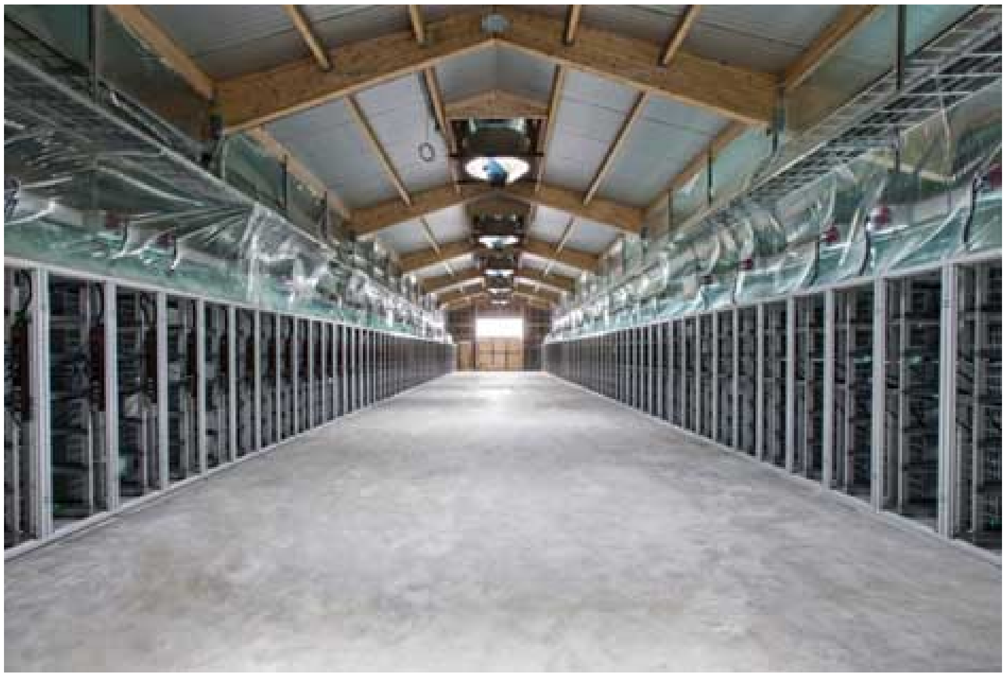


Figure 5.9: BitFury mining center, a professional mining center in the republic of Georgia.

When determining where to set up a mining center, the three biggest considerations are: climate, cost of electricity, and network position. In particular, you need a cool climate so that cooling bills will be kept low. You need cheap electricity, and you need to be well connected to other nodes in the Bitcoin peer-to-peer network so that you can hear about new blocks as they're announced. Georgia and Iceland have been popular destinations for people setting up Bitcoin mining data centers.

Similarities to gold mining. While 'mining' may seem to be just a cute name, if we zoom out a little bit and think about the evolution of mining, we can see really interesting parallels between Bitcoin mining and gold mining. For starters, both of them led to a similar gold rush mentality when initially a lot of young, amateur folks were eager to get into the business.

Whereas with Bitcoin mining we've seen the slow evolution from CPUs to GPUs to FPGAs, to now ASICs, with gold mining we saw the evolution from individuals with gold pans to small groups of people with sluice boxes, to placer mining — which was a big group of people blowing away hillsides with water — to modern gold mining which utilizes a giant open pit to extract tons of raw material from the earth (See Figure 5.10). Both with Bitcoin and with gold, the friendliness and accessibility to individuals has gone down over time and there's been a consolidation with large companies controlling most of the operations.

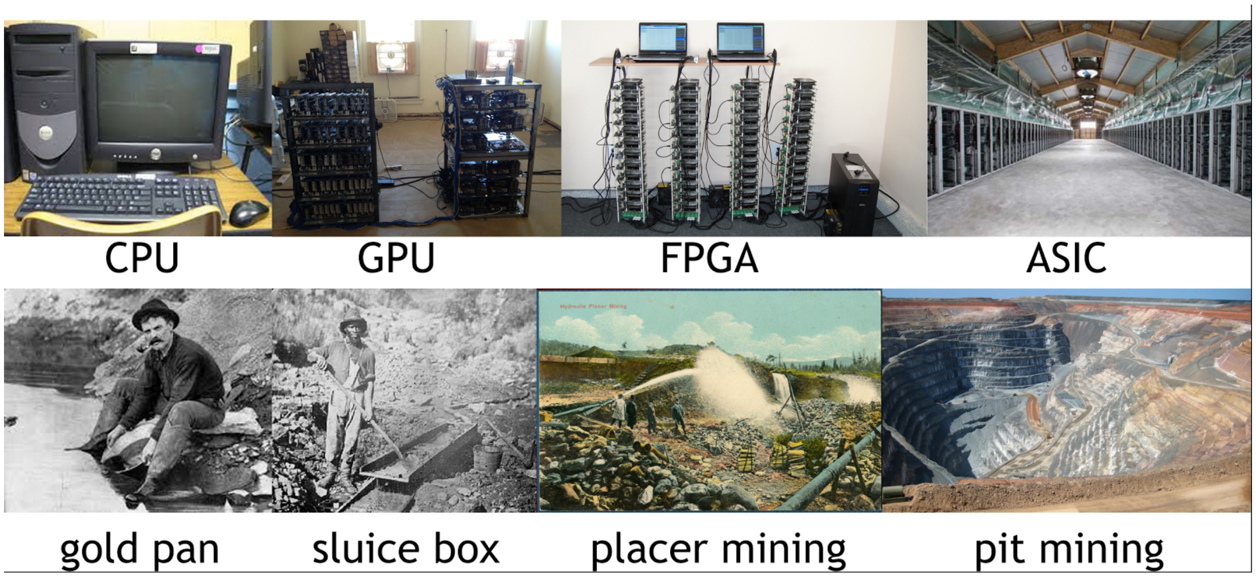


Figure 5.10: Evolution of mining. We can see a clear parallel between the evolution of Bitcoin mining and the evolution of gold mining. Both were initially friendly to individuals and over time became massive operations controlled by large companies.

The future. Currently ASIC mining is the only way to be profitable in Bitcoin and it's not very friendly to small miners. This raises a few questions about what will happen going forward. Are small miners out of Bitcoin mining forever, or is there a way to re-incorporate them? Moreover, does ASIC mining and the development of professional mining centers violate the original vision of Bitcoin which was to have a completely decentralized system in which every individual in the network mined on his or her own computer?

Furthemore, if this is indeed a violation of Satoshi Nakamoto's original vision for Bitcoin, would we be better off with a system in which the only way to mine was with CPUs? In chapter 8, we'll consider these questions and look at alternative forms of mining and how to design mining in a way that is less friendly to ASICs.

5.3 Energy consumption & ecology

We saw how large professional mining data centers have taken over the business of Bitcoin mining, and how this parallels the movement to pit mining in gold mining. You may be aware that a huge concern for environmentalists over the years has been how much damage are these pit mines doing to the environment. Now, Bitcoin is not quite at that level yet, but it is starting to use a significant amount of energy which has become a topic of discussion. In this section we'll see how much energy Bitcoin mining is using and what the implications are for both the currency and the planet.

Thermodynamic limits. There's a physical law known as ***Landauer's principle*** developed by Ralph Landauer in the 1960s that states that any non-reversible computation must use a minimum amount of energy. Logically irreversible computations can be thought of as losses of information. Specifically, the principle states that erasing each bit must consume a minimum of (kT ln 2) joules, where k is the Boltzmann constant (approximately 1.38×10^{-23} J/K), T is the temperature of the circuit in kelvins, and ln 2 is the natural logarithm of 2, roughly 0.69. As you can see, that's an astronomically small amount of energy per bit.

This is derived from basic physics. We're not going to go through the derivation here, but the high-level idea is that every time you flip one bit in a non-reversible way there's a minimum amount of joules that you have to use. Energy is never destroyed; it's converted from one form into another. In the case of computation the energy is mostly transformed from electricity, which is useful, high-grade energy, into heat which is dissipated into the environment.

Now, of course, SHA-256 being a cryptographic hash function is not a reversible computation, and recall from Chapter 1 that this is a basic requirement of cryptographic hash functions. So, since non-reversible computation has to use some energy and SHA-256 — the basis of Bitcoin mining — is not reversible, energy consumption is an inevitable fact of doing Bitcoin mining. That said, the limits placed by Landauer's principle are far, far below the amount of electricity that is being used today. We're nowhere close to the theoretical optimal consumption of computing, but even if we did get to the theoretical optimum we would still be using energy to perform Bitcoin mining.

So why does Bitcoin mining require energy? There are three steps in the process that requires energy, and some of them may not be so obvious.

1. Embodied energy. First, Bitcoin mining equipment needs to be manufactured.This requires physical mining of raw materials as well as turning these raw materials into a Bitcoin mining ASIC, both of which require energy. This is the embodied energy. As soon as you receive a Bitcoin mining ASIC in the mail, you've already consumed a lot of energy — including the shipping energy, of course — before you've even turned it on and tried to mine bitcoins!

Hopefully, over time the embodied energy will go down, as less and less new capacity comes online. As fewer people are going out to buy new mining ASICs, they're going to be obsoleted less quickly, and the embodied energy will be amortized over years and years of mining.

2. Electricity. When your ASIC powered on and mining, it consumes electricity. This is the step that we know has to consume energy due to Landauer's principle. As mining rigs get more efficient, the electrical energy cost will go down. But because of Landauer's principle, we know that it will not disappear; electrical energy consumption will be a fact of life for Bitcoin miners forever.

3. Cooling. A third important component of mining that consumes energy is cooling off your equipment to make sure that it doesn't malfunction. If you’re operating in a very cold climate your cooling cost might be very low, but in most climates you're going to have to pay extra to cool off your equipment from all of the waste heat that it is generating. Generally, the energy used to cool off mining equipment will also be in the form of electricity.

Mining at scale. Both embodied energy and electricity decrease when operating at a large scale. If you're running a large mining data center, you can do it more efficiently. It's cheaper to build chips that are designed to run in a large data center, and you can deliver the power more efficiently as you don't need as many power supplies.

When it comes to cooling, however, the opposite is true. Cooling actually costs mores the larger your scale is. If you want to run a very large operation and have a lot of Bitcoin mining equipment all in one place, there’s less air for the heat to dissipate into in the area surrounding your equipment. Your cooling budget is going to therefore increase because cooling that big mass is going to be much more difficult.

Estimating energy usage. How much energy is the entire Bitcoin system using? Of course, we can't compute this precisely because it's a decentralized network with miners operating all over the place without documenting exactly what they're doing. But there are two basics approaches to estimating how much energy Bitcoin miners are using collectively. We’ll do some back-of-the-envelope calculations here based on early 2015 values. We must emphasize that these figures are very rough, both because some of the parameters are hard to estimate and because they change quickly. At best they should be treated as order-of-magnitude estimates.

Top down approach. The first approach is a top down approach. We start with the simple fact that every time a block is found today 25 bitcoins of rewards, or about 6,500 dollars are given to the miners. That's about 11 dollars every second, being created out of thin air in the Bitcoin economy and given to the miners.
Now let’s ask this question: if the miners are turning all of those 11 dollar per second into electricity, how much can they get? Of course miners aren’t actually spending all of the revenue on electricity, but this will provide an upper bound on the electricity being used. Electricity prices vary greatly, but we’ll estimate that electricity costs around 10 cent per kilowatt-hour (kWh) at an industrial rate in the US, or equivalently 3 cents per megajoule (MJ). If Bitcoin miners were spending all 11 dollars per second of earnings buying electricity, they could purchase 367 megajoules per second, or 367 megawatts (MW).

Sidebar. In the International System of Units (SI), energy is measured in ***joules***. A ***watt*** is a unit of power, where one ***watt*** is defined as one joule per second.

Bottom up approach. A second way to estimate the cost is to use a bottom up approach. In this approach, we look at the number of hashes the miners are actually computing, which we know by observing the difficulty of each block. If we then assume that all miners are using the most efficient hardware, we can derive a lower bound on the electricity consumption.

Currently, the best claimed efficiency figure amongst commercially available mining rigs is about 3 GH/s/W. That is, they can do three billion block hashes per second while consuming 1 watt of power. The total network hashrate is about 350,000,000 GH/s, or equivalently 350 petahertz (PH/s). Multiplying these two together, we see that it takes about 117 MW to produce that many hashes per second at that efficiency. Of course this figure excludes all of the cooling energy and all of the embodied energy that's in those chips, but we're doing an optimal calculation and deriving a lower bound so that's okay.

Combining the top down and bottom up approaches, we can derive a ballpark estimate of the amount of power being used for Bitcoin miners.

You probably don't have a great intuitive sense of how much power this actually is. How much is a megawatt? To build up intuition, let's see how much big power plants produce. One of the largest power plants in the world, the Three Gorges Dam in China is a 10,000 MW power plant. A typical large hydroelectric power plant produces around 1,000 MW. Kashiwazaki-Kariwa, the largest nuclear power plant in the world is a 7,000 MW plant, whereas the average nuclear power plant is about 4,000 MW. A major coal fire plant produces about 2,000 MW.

According to our estimates then, the whole Bitcoin network is consuming maybe 10% of a large power plant's worth of electricity. Although this is not an insignificant amount of power, it's not yet a large amount of electricity compared to all the other things that people are using electricity for on the planet.

Any payment system requires energy and electricity. With traditional currency, lots of energy is consumed guarding and moving gold bullions around, running ATM machines, coin sorting machines, cash registers, and payment processing services, and transporting money in armored cars.

Some people say Bitcoin wastes energy because the energy expended computing SHA-256 hashes doesn't serve any apparent purpose. But you could make this same argument for traditional currency as well — there's a lot of energy being wasted and it doesn't serve any purpose besides maintaining the currency system. So, if we value Bitcoin as a useful currency system, then the energy required to support it is not really being wasted.

Repurposing energy. That said, we can ask if there's a way to do better. One idea is to capture the heat generated from Bitcoin mining do something useful with it instead of just heating up the atmosphere. This is called the data furnaces model. The concept is that instead of buying a traditional electric heater to heat your home, or to heat water in your home, you'd buy a Bitcoin mining rig that you would plug in both to your electricity outlet and also to your Internet connection. Your heater would mine bitcoins and generate heat as a byproduct of that computation. It turns out that the efficiency of doing this isn't much worse than buying an electric heater.

There are a few things about this model that aren't ideal. Although it's about as efficient as using an electric heater, electric heaters are themselves much less efficient than gas heaters. Besides, what happens when everybody turns off their Bitcoin mining rig in the summer? Will mining hash power go down seasonally based on how much heat people need? Will it go way down on days that happen to be warmer than average? This would be really interesting to observe if the data furnace model actually caught on.

The question of ownership is also not clear. If you buy a Bitcoin data furnace, do you own the Bitcoin mining rewards that you get, or does the company that sold them to you? Most people don't have any interest in Bitcoin mining — and probably never will — so it might make more sense to buy it as an appliance and have the company that sold it to you keep the rewards.

Open questions. There are a number of other open questions regarding Bitcoin's energy consumption and its implications.

Will Bitcoin drive out electricity subsidies? In many countries around the world, the government subsidizes electricity — particularly industrial electricity — and one of the reasons they do so is to try to encourage industry to be located in their country. But Bitcoin provides a good way to turn electricity into cash, and this might cause governments to rethink that model. Such subsidies are intended to attract businesses that will contribute to the country's economy, and subsidizing Bitcoin mining arguably doesn't have the intended effect.

Will Bitcoin require people to guard power outlets? Consider universities or corporate building with lots of power outlets. People may try to plug in mining equipment so that they can profit while someone else is paying the electricity bill. In fact, they might use outdated hardware and not bother to upgrade, considering that they will not be paying the electricity bill. Will such locations need security cameras to make sure that people don't plug in Bitcoin mining equipment into un-monitored power outlets and let them run?

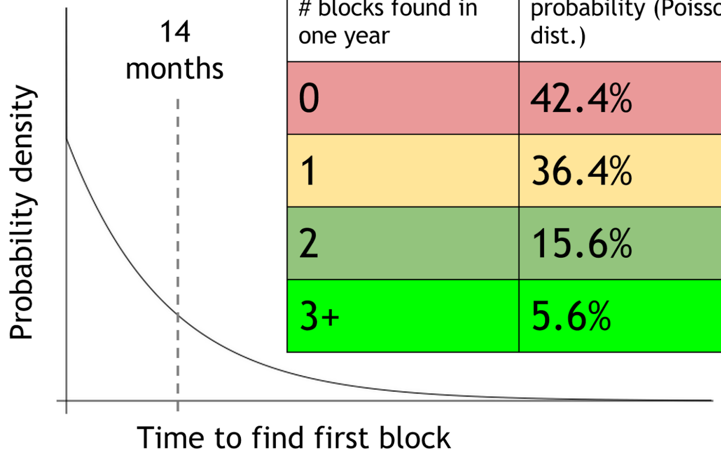
Could we make a currency that didn't have proof of work and didn't have to use so much electricity? This is a question that people are quite interested in, and we'll discuss this in great detail in Chapter 8.

5.4 Mining pools

Consider the economics of being a small miner. Say you're an individual who spent 6,000 dollars of your hard-earned money to buy a nice, shiny, new Bitcoin mining rig. Say that the performance is such that you expect to find a block every 14 months with on average this fancy new rig, and remember that a block is worth about 6,500 dollars as of early 2015.

If you amortize that you could say that the expected revenue of your miner is about 400 dollars per month once you factor in electricity and your other cost of operating it. If you actually got a check in the mail every month for 400 dollars, you'd be quite happy, and it would make a lot of sense to buy the mining rig. But remember that mining is a random process. You don't know when you're going to find the next block. It's a completely random search, and you could find your next block at any time.

High variance. If we look at the distribution of how many blocks you're likely to find in the first year, the variance is pretty high and the expected number of blocks that you'll find is quite low. The distribution is a ***Poisson distribution***, there's a greater than 40% chance that you won't find any blocks within the first year. For an individual miner, this can be devastating. You spent thousands of dollars on the miner, paid lots in electricity to run it, and received nothing in return. There's a roughly 36% chance that you'll find one block within the first year which means maybe you're barely scraping by, provided your electricity costs weren't too high. Finally, there's a smaller chance that you'll find two or more blocks, in which case you could make a nice profit.



# blocks found in one year	probability (Poisson dist.)
0	42.4%
1	36.4%
2	15.6%
3+	5.6%

Figure 5.11: Illustration of uncertainty in mining. We're assuming that the global hash rate is constant and the mean time to find a block is 14 months.

The main point here is that even though on expectation you'll be doing okay — that is, enough to make a return on your money — the variance is sufficiently high that there's a big chance that you'll make nothing at all. For a small miner than, this is essentially a big game of roulette.

Mining pools. Historically, when small business people faced a lot of risk, they formed mutual insurance companies to lower that risk. Farmers, for example, would get together and agree that if any individual farmers barn burnt down they would share profits with that farmer. Can we have a mutual insurance model that works for small Bitcoin miners?

A mining pool is exactly that — mutual insurance for Bitcoin miners. A group of miners will get together, form a pool, and they will all attempt to mine a block with a designated coinbase recipient. That recipient is going to be called the pool manager. So, no matter who actually finds the block, the pool manager will receive the rewards. The pool manager will take that revenue and distribute it to all the participants in the pool based on how much work each participant actually output. Of course, the pool manager will also probably take some kind of cut for their service of managing the pool.

Bitcoin miners lower their variance by joining pools, but how does a pool manager know how much work each member of the pool is actually performing? How can the pool manager divide the revenue commensurate with the amount of work each miner is doing? Obviously the pool manager doesn't want to just take everyone's word for it because people might claim that they've done more than they actually did.

Mining shares. There's an elegant solution to this problem. Miners prove realistically how much work they're doing by outputting shares, or near-valid blocks. Say the target is a number beginning with 67 zeros. The hash must be lower than the target for the block to be valid. In the process of searching for such a block, miners will find blocks with hashes beginning with a lot of zeros, but not quite 67. Miners can show these nearly valid blocks to prove that they are indeed working. A share might require say 40 or 50 zeros, depending on the type of miners the pool is geared for.

```
4AA087F0A52ED2093FA816E53B9B6317F9B8C1227A61F9481AFED67301F2E3FB
D3E51477DCAB108750A5BC9093F6510759CC880BB171A5B77FB4A34ACA27DEDD
0000000008534FF68B98935D090DF5669E3403BD16F1CDFD41CF17D6B474255
BB34ECA3DBB52EFF4B104EBBC0974841EF2F3A59EBBC4474A12F9F595EB81F4B
0000000002F891C1E232F687E41515637F7699EA0F462C2564233FE082BB0AF
0090488133779E7E98177AF1C765CF02D01AB4848DF555533B6C4CFCA201CBA1
460BEFA43B7083E502D36D9D08D64AFB99A100B3B80D4EA4F7B38E18174A0BFB
000000000000000078FB7E1F7E2E4854B8BC71412197EB1448911FA77BAE808A
652F374601D149AC47E01E7776138456181FA4F9D0EEDD8C4FDE3BEF6B1B7ECE
78552640214 3A291CFD60DA09CC80DD066BC723FD5FD20F9B50D614313529AF3
0000000000041EE593434686000AF77F54CDE839A6CE30957B14EDEC10B15C9E5
9C20B06B01A0136F192BD48E0F372A4B9E6BA6ABC36F02FCED22FD9780026A8F
```

Figure 5.12: Mining Shares. Miners continually try to find blocks that hash below the target. In the process, they'll find other blocks whose hashes contain fewer zeros — but still rare enough to prove that they have been working. In this figure, the dull green lines are from shares, while the bright green hash is from a valid block.

Periodically the pool manager will collect transactions and assemble them into a block. The manager will include his or her own address in the coinbase transaction, and send the block to all of the participants in the pool. All pool participants work on this block, and they prove that they've been working on it by sending in shares.

When a member of the pool finds a valid block, he sends it to the pool manager who distributes the reward in proportion to the amount of work done. The miner who actually finds the block is not awarded a special bonus, so if another miner did more work than this miner, that other miner will be paid more. See Figure 5.13.

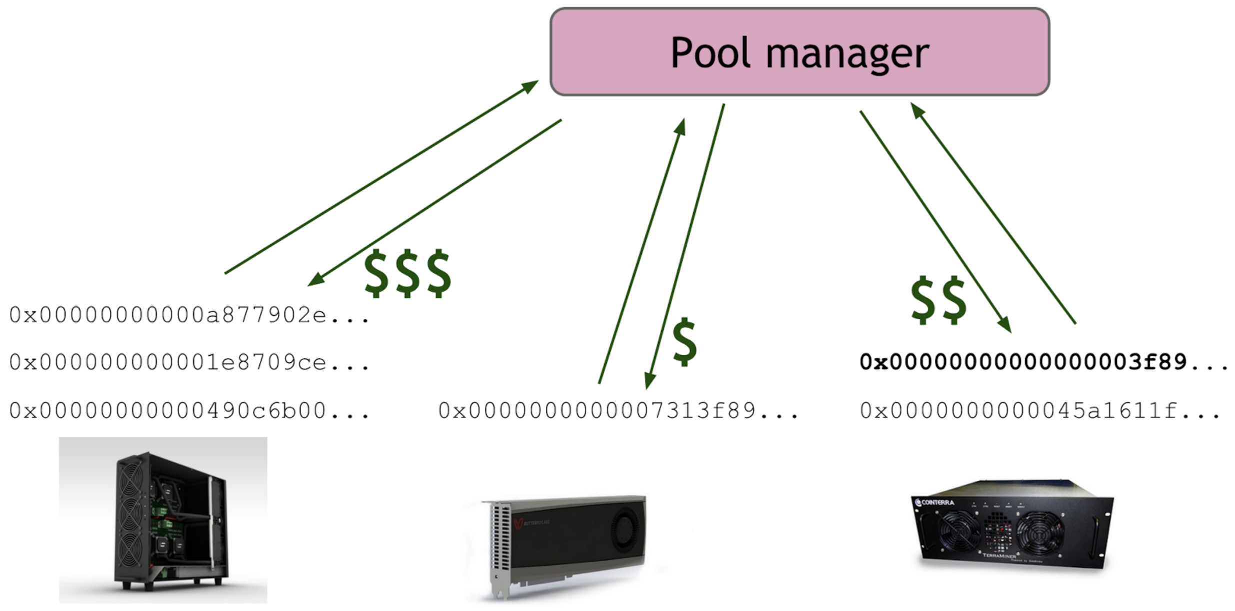


Figure 5.13: Mining rewards. Three participants pictured here are all working on the same block. They are awarded commensurate with the amount of work done. Even though the miner on the right was the one to find the valid block, the miner on the left is paid more since this miner did more work. There is no bonus paid to the miner who actually finds the block.

There are a few options for how exactly the pool manager calculates how much to pay each miner based on the shares they submit. Let’s look at two of the common, simpler ones. There are many others that are also used, but these will illustrate the trade-offs between reward schemes.

Pay-per-share. In the pay per share model, the pool manager pays a flat fee for every share above a certain difficulty for the block that the pool is working on. In this model, miners can send their shares to the pool manager right away and get paid without depending on the pool to find a block.

In some ways, the pay-per-share model is the best for miners. They are guaranteed a certain amount of money every time they find a share. The pool manager essentially absorbs all of the risk since he must pay rewards even if a block is not found. Of course, as a result of the increased risk, in the pay-per-share model, the pool manager will charge higher fees as compared with other models.

One problem with the pay-per-share model is that miners don't actually have any incentive to send valid blocks to the pool manager. That is, they can discard valid blocks, and they will still be paid the same rewards, but will cause a big loss to the pool. A malicious pool manager might attack a competing pool in this fashion to drive them out of business.

Proportional. In the proportional model, instead of paying a flat fee per share, the amount of the share the depends on whether or not the pool actually found a valid block. So every time a valid block is found the rewards from that block are distributed to the members proportional to how much work they actually did.

In the proportional model, the miners still bear some risk proportional to the risk of the pool in general. But if the pool is large enough, the variance of how often the pool finds blocks will be fairly low. Proportional payouts provides lower risk for the pool manager. Proportional mining also gets around the problem that we mentioned with the pay-per-share model. Miners are incentivized to send in the valid blocks that they find because that triggers revenue coming back to them.

The proportional model requires more work on behalf of the pool managers to verify, calculate, and distribute rewards as compared to the flas pay-per-share model.

Mining pools first started around 2010 in the graphics card era or Bitcoin mining. They instantly became very popular for the obvious reason that they lowered the variance for the participating miners. They've become quite advanced now. There are many protocols for how to run mining pools, and it has even been suggested that these mining pool protocols should be standardized as part of Bitcoin itself. That is, just like there's a Bitcoin protocol for running the peer-to-peer network, these protocols are a communication API from the pool manager to all of the members the details of the block to work on, and for the miners to send back to the pool manager the shares that they're finding. Some mining hardware actually supports these protocols at the hardware level. Now this makes it very simple to buy a piece of mining hardware and join a pool. You just plug it into the wall — both the electricity and your network connection — choose a pool, and then it will start immediately getting instructions from the pool, mining and converting your electricity into money.

51% mining pools. As of early 2015, the vast majority of all miners are mining through pools. Very few miners mine on their own anymore. In June 2014, Ghash.io, the largest mining pool, got so big that it actually had over 50% of the entire capacity over the Bitcoin network. Essentially Ghash offered such a good deal to participating miners that the majority wanted to join.

This is something that people had feared for a long time, and there was a backlash against them. By August, Ghash had gone down a little bit, partly by design. Still, two mining pools controlled about half of the power in the network.

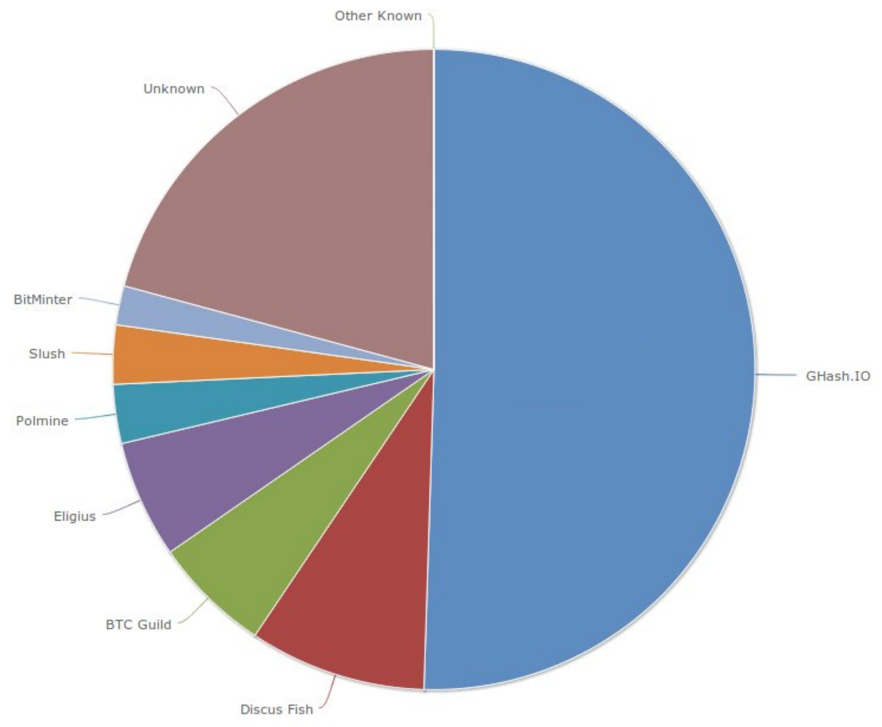


Figure 5.14 (a) Hash power by mining pool, via blockchain.info (June 2014)

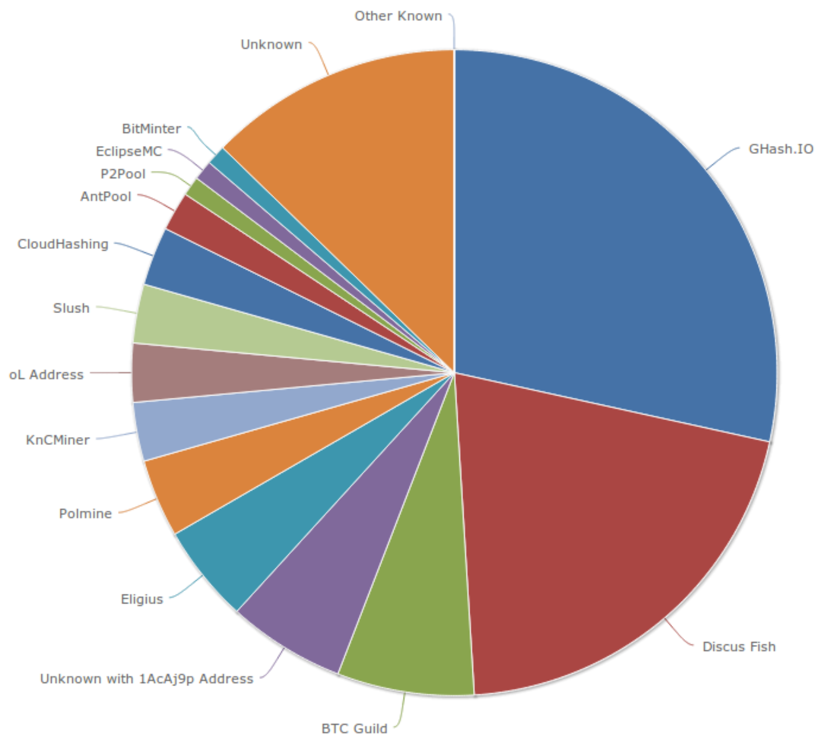


Figure 5.14 (b) Hash power by mining pool, via blockchain.info (August 2014)

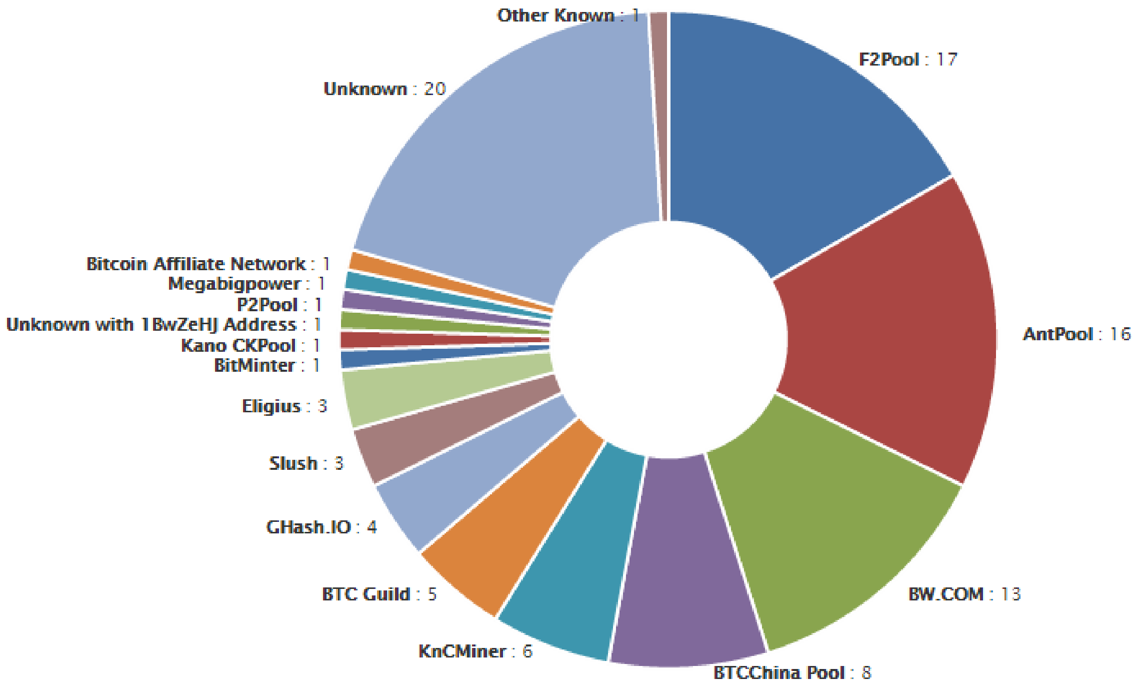


Figure 5.14 (c) Hash power by mining pool, via blockchain.info (April 2015)

In April 2015, the situation looks very different, and less concentrated. The possibility of a pool acquiring 51% is still a concern in the community, but perhaps less so. Due to new miners and pools entering the market and the ease of switching between pools for miners, the market share of different pools remains fluid. It remains to be seen how things will evolve in the long run.

Are mining pools a good thing? The advantages of mining pools are that they make mining much more predictable for the participants and they make it easier for smaller miners to get involved in the game. Without mining pools, the variance would make mining infeasible for small miners.

Another advantage of mining pools is that since there's one central pool manager who is sitting on the network and assembling blocks it makes it easier to upgrade the network. By upgrading the software that the mining pool manager is running that effectively updates all of the software that all the pool members are running.

The main disadvantage of mining pools, of course, is that they lead to centralization. It's an open question how much power the operators of a large mining pool actually have. Of course miners are free in theory to leave a pool if it is perceived as too powerful, but it’s unclear how often miners do so in practice.

Another disadvantage of mining pools is that it lowers the population of people actually running a fully validating Bitcoin node. Previously all miners, no matter how small, had to run their own fully validating node. They all had to store the entire block chain and validate every transaction. Now, most miners offload that task to their pool manager, and this is one reason why as we mention in Chapter 3, the number of fully validated nodes may actually be going down in the Bitcoin network.

If you're concerned about the level of centralization introduced by mining pools, you might ask: could we redesign the mining process so that we don't have any pools and everybody has to mine for themselves? We'll consider this question in Chapter 8.

5.5 Mining incentives and strategies

We've spent most of this chapter describing how the main challenge of being a miner is getting good hardware, finding cheap electricity, getting up and running as fast as you can, and hoping for some good luck. But it turns out that there are also some interesting strategic considerations that every miner has to make before they pick which blocks to work on.

1. *Which transactions to include.* Miners get to choose which transactions they want to include in a block. The default strategy is to include any transaction that includes higher than some minimum transaction fee.
2. *Which block to mine on.* Miners also get to decide on top of which block they want to mine. The default behavior for this decision is to extend the longest valid chain.
3. *Choosing between blocks at the same height.* If two different blocks are mined and announced at around the same time, it results in a 1-block fork, with either block admissible under the longest valid chain policy. Miners then have to decide which block to extend. The default behavior is to build on top of the block that they heard about first.
4. *When to announce new blocks.* When they find a block, miners have to decide when to announce this to the Bitcoin network. The default behavior is to announce it immediately, but they can choose to wait some time before announcing it.

As we see, miners are faced with many decisions. For each decision we mentioned a default strategy. This is the strategy employed by the Bitcoin reference client, which is run by the vast majority of miners at the time of this writing.

But depending on the fraction of mining power controlled by a miner, it may be possible that a non-default strategy is more profitable. Finding such scenarios and strategies is an active area of research. Let’s look at several such potentially profitable deviations. In the following discussion, we’ll assume there’s a deviant miner who controls some fraction of mining power which we’ll denote by α.

Forking attack. The simplest attack is a forking attack, and the obvious way to profit from this attack is to perform a double spend. The miner sends some money to a victim, Bob, in payment for some good or service. Bob waits and sees that the transaction paying him has indeed been included in the block chain (perhaps he follows the common heuristic and even waits for six confirmations to be sure). Convinced that he has been paid, Bob ships the good or performs the service.

The miner now goes ahead and begins working on an earlier block — before the block that contains the transaction to Bob. In this forked chain, the miner inserts an alternate transaction — or a double spend — of the coins paid to Bob back to the miner's own address.

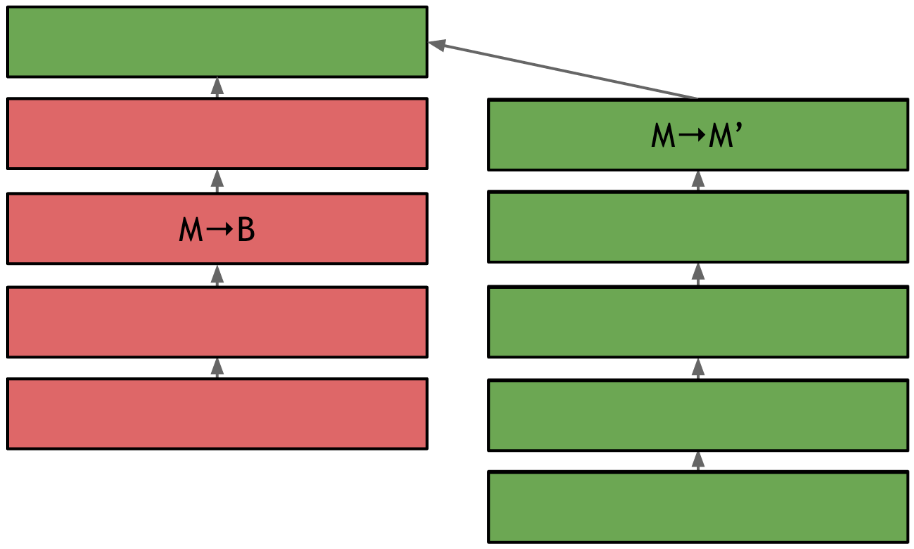


Figure 5.15 Forking attack.

When the miner initially goes back and works on an earlier point in the chain, the doesn't immediately succeed since the forked chain is not the longest chain. However, if the miner has a majority of the hash power — that is, if $\alpha > 0.5$ — the alternate chain eventually become the longest chain, and hence the valid block chain. Once this occurs, the transaction paying Bob no longer exists on the consensus block chain. Moreover, since those coins have already been spent (on the new consensus chain), that transaction can no longer make its way onto the block chain.

Is 51% necessary? Launching a forking attack is certainly possible if $\alpha > 0.5$. In practice, it might be possible to perform this attack with a bit less than that because of other factors like network overhead. The non-attacker chain will have some stale blocks for the usual reason: there is a latency for miners to hear about each others' blocks. But the attacker can avoid most of this latency within his chain. Similarly, the attack gets easier the further over 50 percent you go. People often talk about a 51 percent attacker as if 51% is a magical threshold that suddenly enables a forking attack. In reality, it's more of a gradient.

It's not clear whether a forking attack can actually succeed in practice. The attack is detectable, and it's possible that that the community would decide to reverse the attack by refusing to accept the alternate chain even though it is longer. Moreover, it's possible that such an attack occurring would completely crash the Bitcoin exchange rate. If a miner carried out such an attack, people might lose confidence in the system and refrain from buying bitcoins causing the exchange rate to fall.

For these reasons, the most likely motivation for a forking attack is to destroy the currency by a dramatic loss of confidence. This has been referred to as a Goldfinger attack after the Bond villain that tried to irradiate all the gold in Fort Knox to make it valueless. A Goldfinger attacker’s goal might be to destroy the currency, or possibly to profit by either having shorted Bitcoin having significant holdings in some competing currency.

Forking attack via bribery. Buying enough hardware to control the majority of the hash power seems like quite an expensive and difficult task. But's possible that there is an easier way to launch a forking attack. Whereas it would be really expensive to buy enough mining capacity to have more than everybody else in the world, it might be possible to bribe the people who do control all that capacity to work on your behalf

There are a few ways that you could bribe miners. One way is to do this “out of band” — perhaps locate some large miners and hand them an envelope of cash for working on your chain. A more clever technique is to declare yourself to be a new mining pool and run it at a loss. You could offer greater incentives than other pools and cause many miners to join your pool. Even though the incentives you offer will not be sustainable, you may be able to keep them going for long enough to successfully launch a forking attack and perhaps profit. A third technique for bribing is to leave big tips in your forking blocks — big enough to cause miners to leave the longest chain and work on your chain in hopes that it will become the longest chain and they will collect the tip.

However you actually go about doing the bribing, the idea is the same: instead of actually acquiring all the mining capacity yourself, you just pay the people who already have it to work on your fork.

Perhaps miners won’t help out your attack because to do so would hurt the currency in which they have invested so much money and mining equipment. On the other hand, while miners as a group might want to keep the currency solvent, they don’t act collectively. Individual miners might defect and accept a bribe if they thought they could make more money in the short term. This would be a classic tragedy of the commons from an economic perspective.

None of this has actually happened. It's an open question if a bribery attack like this could actually be viable.

Block-withholding attacks. Say that you just found a block. The default behavior is to immediately announce it to the network, but if you’re carrying out a block-withholding attack, you do not announce it right away. Instead you try to get ahead by doing some more mining on top of this block in hopes of finding two blocks in a row before the rest of network finds even one, and you keep these blocks to yourself as a secret.

If you’re ahead of the public block chain by two blocks, all of the mining effort of the rest of the network will be wasted. Other miners will mine on top of what they think is the longest chain, but as soon as they find a valid block, you can announce the two blocks that you were withholding. That would instantly be the new longest valid chain and the block that the rest of the network worked so hard to find would immediately be orphaned, and cut off from the longest chain. This is known as ***selfish mining.*** By wasting some of the hash power of the rest of the network, you hope to increase your effective share of mining rewards.

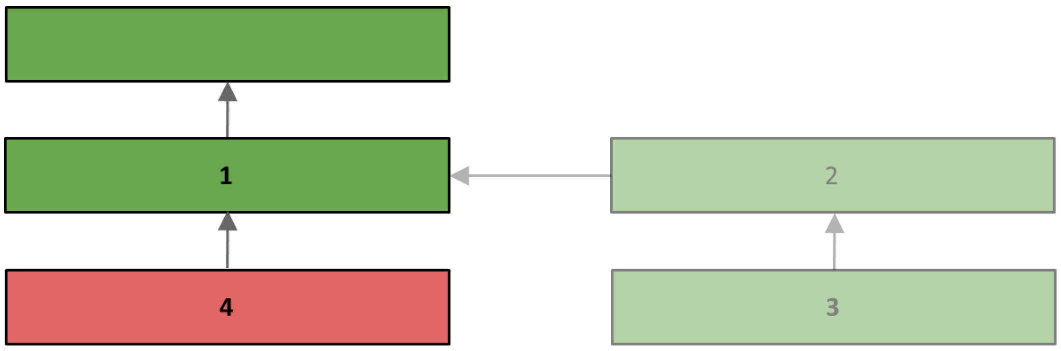


Figure 5.16: Illustration of selfish mining. This shows one of several possible ways in which the attack could play out. (1) Block chain before attack. (2) Attacker mines a block, withholds it, starts mining on top of it. (3) Attacker gets lucky, finds a second block before the rest of the network, continues to withhold blocks. (4) Non-attacker finds a block and broadcasts it. In response, the attacker broadcasts both his blocks, orphaning the red block and laying waste the mining power that went into finding it.

The problem is that you need to get lucky to find two blocks in a row. Chances are that someone else in the network announces a valid block when you’re only one block ahead. If this happens, you'll want to immediately announce your secret block yourself. This creates a 1-block fork and every miner will need to make a decision about which of those blocks to mine on. Your hope is that a large fraction of other miners will hear about your block first and decide to work on it. The viability of this block withholding approach is going to depend very heavily on your ability to win these races. So your network position is of key importance here. You could try to peer with every node so that your block will reach most nodes first.

As it turns out, if you assume that you only have a 50 percent chance of winning these races, selfish mining is an improvement over the default strategy if $\alpha > .25$. The existence of this attack is quite surprising, and it's contrary to the original widely-held belief that without a majority of the network — that is with $\alpha \leq .5$, there was no better mining strategy then the default. So it's not safe to assume that a miner who doesn't control 50 percent of the network doesn't have anything to gain by switching to an alternate strategy.

At this point selfish mining is just a theoretical attack and hasn't been observed in practice. Selfish mining would pretty easy to detect because it would increase the rate of near-simultaneous block announcements.

Blacklisting and punitive forking. Say a miner wants to blacklist transactions from address *X*. In other words, they want to freeze the the money held by that address making it unspendable. Perhaps you intend to profit off of this by some sort of ransom or extortion demanding that the person you're blacklisting pay you in order to be taken off of your blacklist. Blacklisting also might be something that you want to do for legal reasons. Maybe certain addresses are designated as bad by the government, and law enforcement may demand that all miners operating in their jurisdiction blacklist those addresses.

The traditional wisdom is that there's no effective way to blacklist addresses in Bitcoin. Even if some miners refuses to include some transactions in blocks, other miners will. If you're a miner trying to blacklist, you could try something stronger, namely, punitive forking. You could announce that you'll refuse to work on a chain containing a transaction originating from this address. This is quite an extreme strategy if you have less than the majority of the network hash power. By announcing that you'll refuse to mine on any chain that has certain transactions, if such a chain does come into existence and is accepted by the rest of the network as the longest chain, you will have cut yourself off from the consensus chain forever, and all of the mining that you're doing is essentially wasted.

Feather-forking. In other words, a threat to blacklist certain transactions via punitive forking in the above manner is not credible as far as the other miners are concerned. But there's a much more clever way to do it. Instead of announcing that you're going to fork forever as soon as you seen an a transaction originating from address *X*, you announce that you're going to fork if you see a block that has a transaction from address *X*, but you will give up after a while — typically after one or two blocks confirm the transaction from address *X,* you'll go back to the longest chain.

If you give up after one confirmation, your chance of orphaning the block with the transaction from *X* is α^2. The reason for this is that you'll have to find two consecutive blocks to get rid of the block with the transaction from address *X* before the rest of the network finds a block, and α^2 is the chance that you will get lucky twice.

A chance of α^2 might not seem very good. If you control 20% of the hash power, there's only a 4% chance of actually getting rid of that transaction that you don't want to see in the block chain. But it's better than it might seems as you might motivate other miners to join you. As long as you've been very public about this, other miners know that if they include a transaction from address *X*, they have an α^2 chance that the block that they find will end up being orphaned because of your feather-forking attack. If they don't have any strong motivation to include that transaction from address X and it doesn't have a high transaction fee, the α^2 chance of losing their mining reward might be a much bigger incentive than collecting the transaction fee.

It emerges then that other miners may rationally decide to join you in enforcing the blacklist, and you can therefore enforce a blacklist even if $\alpha < .5$. If you have less than a majority of the mining capacity, the success of this attack is going to depend heavily on how convincing you are to the other miners that you're definitely going to fork.

Transitioning to mining rewards dominated by transaction fees. As of 2015 transaction fees don't matter that much since block rewards provide the vast majority — well over 99% — of all the revenue that miners are making. But every four years the block reward is cut in half, and eventually, the block reward will be low enough that transactions fees are going to be the main source of revenue for miners. It's an open question as to how exactly miners will operate when transaction fees become their main source of income. Are miners going to be more aggressive in enforcing minimum transaction fees, and how are they going to cooperate to enforce that?

In summary, miners are free to implement any strategy that they want although in practice we've seen very little behavior of anything other than the default strategy. There's no complete model for miner behavior that says that entire default strategy is optimal, and in this chapter we've seen specific examples of deviations that may be profitable for miners with sufficient hash power. Mining strategies is an area in which the practice is ahead of the theory. Empirically, we've seen that in a world where most miners do choose the default strategy, Bitcoin seems to work well. But we're not sure if it works in theory yet.

We also can't be sure that it will always continue to work well in practice. The facts on the ground are going to change for Bitcoin. Miners are becoming more centralized and more professional, and the network capacity is increasing. Besides, in the long run Bitcoin must contend with the transition from fixed mining rewards to transaction fees. We don't really know how this will play out, and using game-theoretic models to try to predict it is a very interesting current area of research.

Further reading

An excellent paper on the evolution of mining hardware:

Taylor, Michael Bedford. Bitcoin and the age of bespoke Silicon. Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems. IEEE Press, 2013.

The "systematization of knowledge" paper on Bitcoin and cryptocurrencies, especially Section III on Stability:

Bonneau, Joseph, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. Proceedings of 2015 IEEE Security and Privacy Conference, 2015.

A comprehensive 2011 paper analyzing different reward systems for pooled mining (some of the information is a bit out of date, but overall it’s still a good resource):

Rosenfeld, Meni. Analysis of bitcoin pooled mining reward systems. arXiv preprint arXiv:1112.4980 (2011).

Several papers that analyze mining strategy:

Eyal, Ittay, and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. Financial Cryptography and Data Security. Springer Berlin Heidelberg, 2014.

Kroll, Joshua A., Ian C. Davey, and Edward W. Felten. The economics of Bitcoin mining, or Bitcoin in the presence of adversaries. Proceedings of WEIS. Vol. 2013.

Eyal, Ittay. The Miner's Dilemma. Proceedings of 2015 IEEE Security and Privacy Conference, 2015.

Bitcoin and Cryptocurrency Technologies

Arvind Narayanan, Joseph Bonneau, Edward Felten,
Andrew Miller, Steven Goldfeder

Draft — May 8, 2015

Feedback welcome! Email bitcoinbook@lists.cs.princeton.edu

Chapter 7: Community, Politics, and Regulation

In this chapter we'll look at about all the ways that the world of Bitcoin and cryptocurrency technology touches the world of people. We'll discuss the community, politics within Bitcoin and the way that Bitcoin interacts with politics, and law enforcement and regulation issues.

7.1: Consensus in Bitcoin

First let's look at consensus in Bitcoin, that is, the way that the operation of Bitcoin relies on the formation of consensus amongst people. There are three kinds of consensus that have to operate for Bitcoin to be successful.

1. Consensus about rules. By rules we mean things like what makes a transaction valid, what makes a block valid, and how the nodes in the peer-to-peer network should behave — how they should interact with each other, the communication protocol they should use, and more generally all the protocols and data formats that are involved in making Bitcoin work.

You need to have a consensus about these things so that all the different participants in the system can talk to each other and agree on what's happening.

2. Consensus about history. That is, consensus about what's in and what isn't in the block chain, and therefore a consensus about which transactions have occurred. Once you have that, what follows is a consensus about which coins — which unspent outputs — exist and who owns them.

This consensus results from the processes we've looked at in earlier chapters from which the block chain is built and by which nodes come to consensus about the contents of the block chain. This is the most familiar and most technically intricate kind of consensus in Bitcoin.

3. Consensus that coins are valuable. The third form of consensus is the general agreement that bitcoins are valuable, that bitcoins are a good thing to have, and in particular the consensus that if someone gives you a bitcoin today, then tomorrow you will be able to redeem or trade that for something of value.

Any currency needs this — whether it's a fiat currency like the dollar or cryptocurrency like Bitcoin, you need a consensus that the thing has value. That is, you need people to generally accept that it's exchangeable for something of value, now and in the future. In a fiat currency, this is the *only* kind of consensus, whereas in cryptocurrencies we additionally have the first two.

In Bitcoin, this form of consensus, unlike the others, is a bit circular. In other words, my belief that the bitcoins I'm receiving today are of value depends on my expectation that tomorrow other people will believe the same thing. So consensus on value relies on believing that consensus on value will continue. This is sometimes called the Tinkerbell effect by analogy to Peter Pan where it's said that Tinker Bell exists because you believe in her.

Whether it's circular or not, it seems to exist and it's important for Bitcoin to operate. Now, what's important about all three forms of consensus is that they're intertwined with each other, as Figure 7.1 shows.

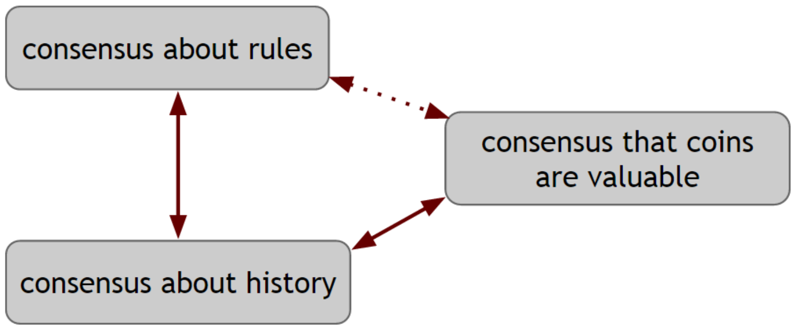


Figure 7.1: Relationships between the three forms of consensus in Bitcoin

First of all, consensus about rules and consensus about history go together. Without knowing which blocks are valid you can’t have consensus about the block chain. And without consensus about which blocks are in the block chain, you can’t know if a transaction is valid or if it’s trying to spend an already-spent output.

Consensus about history and consensus that coins are valuable are also tied together. Consensus about history means that we agree on who owns which coins, and that’s a prerequisite for believing that the coins have value — without a consensus that I own a particular coin I can’t have any expectation that people will accept that coin from me as payment in the future. It’s true in reverse as well — as we saw in Chapter 2, consensus about value is what incentivizes miners to maintain the security of the block chain, which gets us consensus about history.

The genius in Bitcoin’s original design was in recognizing that it would be very difficult to get any one of these types of consensus by itself. Consensus about the rules in a worldwide decentralized environment where there's no notion of identity isn’t the kind of thing that's likely to happen.

Consensus about a history, similarly, is a very difficult distributed data structure problem which is not likely to be solvable on it's own. And a consensus that some kind of cryptocurrency has value is also very difficult to achieve. What the design of Bitcoin and the continued operation of Bitcoin shows is that even if you can't build any one of these forms of consensus by itself you can somehow stand up all three of them together, and get them to operate in an interdependent way. So when we talk about how things operate in the Bitcoin community we have to bear in mind that Bitcoin relies on agreement by the participants, and that consensus is a fragile and interdependent thing.

7.2: Bitcoin Core Software

The Bitcoin Core software is a piece of open-source software which is a focal point for discussion and debate about Bitcoin's rules.

Bitcoin Core is licensed under the MIT license which is a very permissive open-source license. It allows the software to be used for almost any purpose as long as the source is attributed and the MIT license is not stripped out. Bitcoin Core is the most widely used Bitcoin software, and even those who don't use it tend to look to it to define what the rules are. That is, people building alternative Bitcoin software typically try to mimic the rule-defining parts of the Bitcoin Core software, the parts that check validity of transactions and blocks.

Bitcoin Core is the de-facto rulebook of Bitcoin. If you want to know what's valid in Bitcoin, the Bitcoin Core software — or explanations of it — is where to look.

Bitcoin Improvement Proposals. Anyone can contribute technical improvements via "pull requests" to Bitcoin Core, a familiar process in the world of open-source software. For more substantial changes, especially protocol modifications, there is a process called Bitcoin Improvement Proposals or BIPs. These are formal proposals for changes to Bitcoin. Typically a BIP will include a technical specification for a proposed change as well as a rationale for it. So if you have an idea for how to improve Bitcoin by making some technical change, you're encouraged to write up one of these documents and to publish it as part of the Bitcoin Improvement Proposal series, and that will then kick off a discussion in the community about what to do. While the formal process is open to anyone, there's a learning curve for participation like any open-source project.

BIPs are published in a numbered series. Each one has a champion, that is, an author who evangelizes in favor of it, coordinates discussion and tries to build a consensus within the community in favor of going forward with or implementing a particular proposal.

What we said above applies to proposals to change the technology. There are also some BIPs that are purely informational and exist just to tell people things that they might not otherwise know, or that are process oriented, that talk about how things should be decided in the Bitcoin community.

In summary, Bitcoin has a rulebook as well as a process for proposing, specifying, and discussing rule changes, namely BIPs.

Bitcoin Core developers. To understand the role of the Bitcoin Core software we also have to understand the role of Bitcoin Core developers. The original code was written by Satoshi Nakamoto, who we'll return to later in the chapter. Nakamoto is no longer active, but instead there are a group of developers who maintain Bitcoin Core. As of early 2015 there are five: Gavin Andresen, Jeff Garzik, Gregory Maxwell, Wladimir J. van der Laan, and Pieter Wuille. The Core developers lead the effort to continue development of the software and are in charge of which code gets pushed into new versions of Bitcoin Core.

How powerful are these people? In one sense they’re very powerful, because you could argue that any the rule changes to the code that they make will get shipped in Bitcoin Core and will be followed by default. These are the people who hold the pen that can write things into the de-facto rulebook of Bitcoin. In another sense, they’re not powerful at all. Because it's open-source software, anyone can copy it and modify it, in other words, fork the software at any time, and so if the lead developers start behaving in a way that the community doesn't like, and strongly rejects, the community can go in a different direction.

One way of thinking about this is to say that the lead developers are leading the parade. They’re out in front of the parade marching and the parade will generally follow them when they turn a corner, but if they try to lead the parade into an action that disastrous, then the parade members marching behind them might decide to go in a different direction. They can urge people on, and as long as they seem to be behaving reasonably, the group will probably follow them, but they don't have formal power to force people to follow them if they take the system in a technical direction that the community doesn't like.

Let’s think about what you as a user of a system can do if you don't like the way the rules are going or the way it's being run, and compare it to a centralized currency like a fiat currency. In a centralized currency if you don't like what's going on you have a right to exit, that is, you can stop using it. First you’d have to try and sell any currency you hold. Just like almost any business that you deal with, you have the ability to just stop dealing with them if you don't like what they're doing. On the other hand, if it's a currency and you've got a lot of business, you've got a lot of assets tied up in it and it might be expensive or difficult to actually exit. Whether or not it’s easy, with a centralized currency that's really your only option.

With Bitcoin, while you certainly have the right to exit, because it operates in an open-source way, you additionally have the right to fork the rules. That means you, and some of your friends and colleagues can decide that you would rather live under a different rule set, and you can fork the rules and go a different direction from the lead developers. The right to fork this is more empowering for users than the right to exit, and therefore the community has more power in a system like Bitcoin which is open source than it would in a purely centralized system. So although the lead developers might look like a centralized entity controlling things, in fact they don't have the power that a purely centralized manager or software owner would have.

Forks in the rules. One way to fork the software and the rules is to start a new block chain with a new genesis block. This is popular option for creating altcoins, and we’ll discuss altcoins in Chapter 10. But for now let’s consider a different type of fork in the rules, one in which those who fork decide to fork the block chain as well.

If you recall the distinction between a hard fork and a soft fork from Chapter 3, we're talking about a hard fork here. At the point when there's a disagreement about the rules, there will be a fork in the block chain, resulting in two branches. One branch is valid under rule set A but invalid under rule set B, and vice versa. Once the miners operating under the two rule sets separate they can't come back together because each branch will contain transactions or blocks that's invalid according to the other rule set.

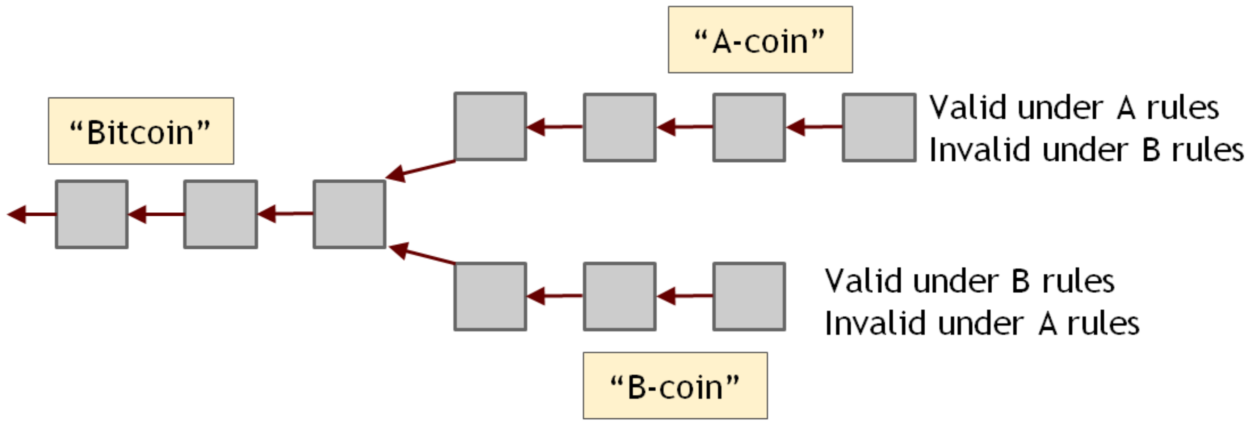


Figure 7.2: A fork in the currency. If a fork in the rules leads to a hard fork in the block chain, the currency itself forks and two new currencies result.

We can think of the currency we had up until the fork as being Bitcoin — the big happy Bitcoin that everyone agreed on. After the fork it's as if there are two new currencies which we can think of as being A-coin corresponding rule set A and B-coin corresponding to rule set B. At the moment of the fork, it's as if everyone who owned one Bitcoin receives one A-coin and one B-coin. From that point on, A-coin and B-coin will operate separately as if they were separate currencies, and they might operate independently. The two groups might continue to evolve their rules in different ways.

We should emphasize that it's not just the software, or the rules, or the software implementing the rules that forked — it's the currency itself that forked. This is an interesting thing that can happen in a cryptocurrency that couldn't happen in a traditional currency where the option of forking is not available to users. To our knowledge, neither Bitcoin nor any altcoin has ever forked in this way, but it's a fascinating possibility.

How might people respond to a fork like this? It depends on why the fork happened. The first case is where the fork was not intended as a disagreement about the rules, but instead as a way of starting an altcoin. Someone might start an altcoin by forking Bitcoin's block chain if they want to start with a ruleset that's very close to Bitcoin's. This doesn't really pose a problem for the community — the altcoin goes its separate way, the branches coexist peacefully, and some people will prefer to use bitcoins while others will prefer the altcoin. But as we said earlier, as far as we know, no one's ever started an altcoin by forking Bitcoin's or another existing altcoin's block chain. They've always started with a new genesis block.

The interesting case is if the fork reflected a fight between two groups about what the future of Bitcoin should be — in other words, a rebellion within the Bitcoin community where a sub-group decides to break off and decides they have a better idea about how the system should be run. In that case, the two branches are rivals and will fight for market share. A-coin and there's a B-coin will each try to get more merchants to accept it and more people to buy it. Each will want to be perceived as the "real Bitcoin." There may be a public-relations fight where each claims legitimacy and portrays the other as a weird splinter group.

The probable outcome is that one branch will eventually win and the other will melt away. These sorts of competitions tend to tip in one direction. Once one of the two gets seen as more legitimate and obtains a bigger market share, the network effect will prevail and the other becomes a niche currency and will eventually fall away. The rule set and the governance structure of the winner will become the de-facto rule set and governance structure of Bitcoin.

7.3: Stakeholders: Who's in Charge?

Who're the stakeholders in Bitcoin, and who's really in charge? We've seen how Bitcoin relies on consensus and how its rulebook is written in practice. We've analyzed the possibility of a fork or a fight about what the rules should be. Now let's take up the question of who has the power to determine who might win a fight like that.

In other words, if there's a discussion and negotiation in the community about rule-setting, and that negotiation fails, we want to know what will determine the outcome. Generally speaking, in any negotiation, the party that has the best alternative to a negotiated agreement has the advantage in a negotiation. So figuring out who might win a fight will tell us who has the upper hand in community discussions and negotiations about the future of Bitcoin.

We can make a bunch of different claims on behalf of different stakeholders.

1. Core developers have the power — they write the rulebook and almost everybody uses their code.
2. Miners have the power — they write history and decide which transactions are valid. If miners decide to follow a certain set of rules, arguably everyone else has to follow it. The fork with more mining power behind it will build a stronger, more secure block chain and so has some ability to push the rules in a particular direction. Just how much power they have depends on whether it's a hard fork or a soft fork, but either way they have some power.
3. Investors have the power — they buy and hold bitcoins, so it's the investors who decide whether Bitcoin has any value. You could argue that if the developers control consensus about the rules and the miners control consensus about history, it's the investors who control consensus that Bitcoin has value. In the case of a hard fork, if investors mostly decide to put their money either A-coin or B-coin, that branch will be perceived as legitimate.

4. Merchants and their customers have the power — they generate the primary demand for Bitcoin. While investors provide some of the demand that supports the price of the currency, the primary demand that drives the price of the currency, as we saw in Chapter 4, arises from a desire to mediate transactions using Bitcoin as a payment technology. Investors, according to this argument, are just guessing where the primary demand will be in the future.
5. Payment services have the power — they're the ones that handle transactions. A lot of merchants don't care which currency they follow and simply want to use a payment service that will give them dollars at the end of the day, allow their customers to pay using a cryptocurrency, and handle all the risk. So maybe payment services drive primary demand and merchants, customers, and investors will follow them.

As you may have guessed, there's some merit to all these arguments, and all of those entities have some power. In order to succeed, a coin needs all these forms of consensus — a stable rulebook written by developers, mining power, investment, participation by merchants and customers, and the payment services that support them. So all of these parties have some power in controlling the outcome about a fight over the future of Bitcoin, and there's no one that we can point to as being the definite winner. It's a big, ugly, messy consensus-building exercise.

The Bitcoin Foundation. There's one more player that's relevant to the governance of Bitcoin and that's the Bitcoin Foundation. The Bitcoin Foundation was founded in 2012 as a nonprofit. It's played two main roles. The first is funding some of the Core developers out of the foundation's assets so that they can work full time on continuing to develop the software. The second is talking to government, especially the US government, as the "voice of Bitcoin."

Now, some members of the Bitcoin community believe that Bitcoin should operate outside of and apart from traditional national governments. That believe Bitcoin should operate across borders and shouldn't explain or justify itself to governments or negotiate with them. Others take a different view. They view regulation as inevitable, desirable, or both, and would like the interests of the Bitcoin community to be represented in government, and for the community's arguments to be heard. The Foundation arose partly to fill this need, and it's fair to say that its dealings with government have done a lot to smooth the road for an understanding and acceptance of Bitcoin.

The Foundation has had quite a bit of controversy. Some board members have gotten into criminal or financial trouble, and there have been questions about the extent to which some of them represent the community. The Foundation has had to struggle with members of the board that become liabilities and have to be replaced on short notice. It's been accused of lacking transparency and of being effectively bankrupt. As of early 2015, it's at best unclear if the Bitcoin Foundation will have much of a role in Bitcoin's future.

A different non-profit group, Coin Center, launched in September 2014 based in Washington, D.C., has taken on one of the roles the Bitcoin Foundation played, namely advocacy and talking to government. Coin Center acts as a "think tank." It has operated without much controversy as of early 2015. Neither the Bitcoin Foundation nor Coin Center is in charge of Bitcoin anymore than any of the other stakeholders. The success and perceived legitimacy of any such representative entity will be driven by how much support — and funding — it can obtain from the community over time, like everything else in this kind of open source based ecosystem.

To summarize, there's no one entity or group that is definitively in control of Bitcoin's evolution. In another sense, everybody is in charge because it's the existence of consensus about how the system will operate — the three interlocking forms of consensus, on rules, on history, and on value — that governs Bitcoin. Any ruleset, group, or governance structure that can maintain that consensus over time will, in a very real sense, be in charge of Bitcoin.

7.4: Roots of Bitcoin

Let's look at the roots of Bitcoin — how it got started, what its precursors were, and what we know about its mysterious founder.

Cypherpunk and digital cash. There are two precursors to Bitcoin worth discussing. One of these was *cypherpunk*, a movement that brought together two viewpoints. First was libertarianism and in particular the idea that society would be better off with either no government or very minimal government. Together with that strong libertarian notion or perhaps even anarchist notion, we had the idea of strong cryptography and in particular public-key cryptography which started in the late 1970s. The cypherpunk movement was a group of people who believed that with strong online privacy and strong cryptography you could re-architect the way that people interact with each other. In this world, cypherpunks believed, people could protect themselves and their interests more effectively and with much less activity (or, as they would say, interference) from government.

One of the challenges in the cypherpunk movement was how to deal with money in a future cypherpunk world where people were interacting online via strong technical and cryptographic measures. In response, a bunch of research came along, led especially by early digital cash work by David Chaum and others, that was designed to create new forms of digital value that functioned like money, specifically cash, in the sense of being anonymous and easily exchangeable. There's a whole interesting story about how these technical ideas were developed and why early digital cash *didn't* sweep the world, but we won't go into it here. In any event, early work in that area came together with cypherpunk beliefs and in particular the desire to have a strong currency that would be decentralized, online, and relatively private to sow the seeds from which Bitcoin would be born. It's also the basis for the philosophy that many of Bitcoin's supporters follow.

Satoshi Nakamoto. Bitcoin began in 2008 with the release of this white paper called *Bitcoin: A Peer to Peer Electronic Cash System* that was authored by Satoshi Nakamoto. This paper, which you can find online easily, is the initial description of what Bitcoin is, how it works, and the philosophy behind its design. It's still a good resource to get a quick idea of how Bitcoin's technical design and philosophy were specified. Open-source software implementing that specification was released soon after by the same Satoshi Nakamoto, and that’s where everything started. To this day, Satoshi is one of the central mysteries of Bitcoin.

We know that the name Satoshi Nakamoto is almost certainly a pseudonym. It’s a fake name that some person or people have adopted for the purpose of doing things related to Bitcoin. The identity of Satoshi is associated with certain public keys, certain accounts and certain systems. That means there are certain online activities or digital signatures that would convince the community that something was said by or issued by or created by the real Satoshi. So Satoshi, while being a pseudonym, is also a person (or people) who can speak, and who has spoken especially extensively in the early history of Bitcoin. Satoshi was fairly active in working on and writing about Bitcoin, and participating in online forums until around 2010, and since that time Satoshi has said almost nothing.

We know that Satoshi writes fairly well in English. Satoshi uses sometimes American and sometimes British spellings. There have been numerous attempts to look at Satoshi’s text, code, post times, machine identifiers, and so on to try to answer questions like: what is Satoshi's native language? Where is Satoshi from? The real identity of Satoshi is still unknown, despite occasional confident pronouncements by individuals and, at least once, a news organization.

Satoshi owns a lot of bitcoins from early mining. In the beginning Satoshi was perhaps the only miner, or one of the only few people mining bitcoins. So until Bitcoin mining took off and the network’s hash rate started to increase from the influx of other miners, Satoshi was accumulating all or at least a significant portion of block rewards, which was 50 bitcoins every 10 minutes. As Bitcoin’s price appreciated, this turned into a large sum of wealth. We know that these bitcoins haven’t been cashed out. Everybody can see which Bitcoin addresses probably belong to Satoshi, and so if those coins were to be sold and the proceeds transferred into any particular bank account, it would be a very notable event and an important clue to Satoshi's identity. So, interestingly, even though Satoshi has on paper made a lot of profit from Bitcoin mining, Satoshi is unable to cash in that profit without identifying himself or herself, and that’s something that, for whatever reason, Satoshi doesn't want to do.

In an important sense it doesn't matter that we don’t know Satoshi’s identity because of the notable feature of Bitcoin that it is decentralized and with no single entity in charge. Satoshi's not in charge, and to some extent it doesn't really matter what Satoshi thinks anymore. Any special influence that Satoshi has is only because of respect that Satoshi would have in the Bitcoin community should Satoshi become active again.

Growth. Bitcoin has grown a lot since the system became operational in January 2009. We can see it in the graph of transaction volume (Figure 7.3) and in the graph of exchange rate (7.4), although the peak price, as of April 2015, was back in late 2013. Sometimes the growth has been gradual, but sometimes there have been jumps or spurts, often corresponding to newsworthy events. Generally speaking, the growth has accelerated over time.

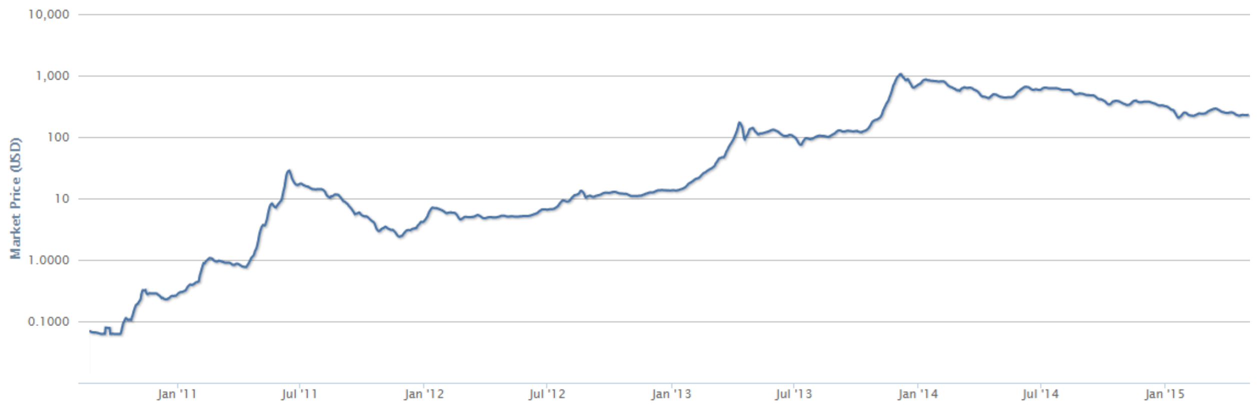


Figure 7.3: Market Price of Bitcoin (7-day average). Note the logarithmic scale.
Source: bitcoincharts.com.

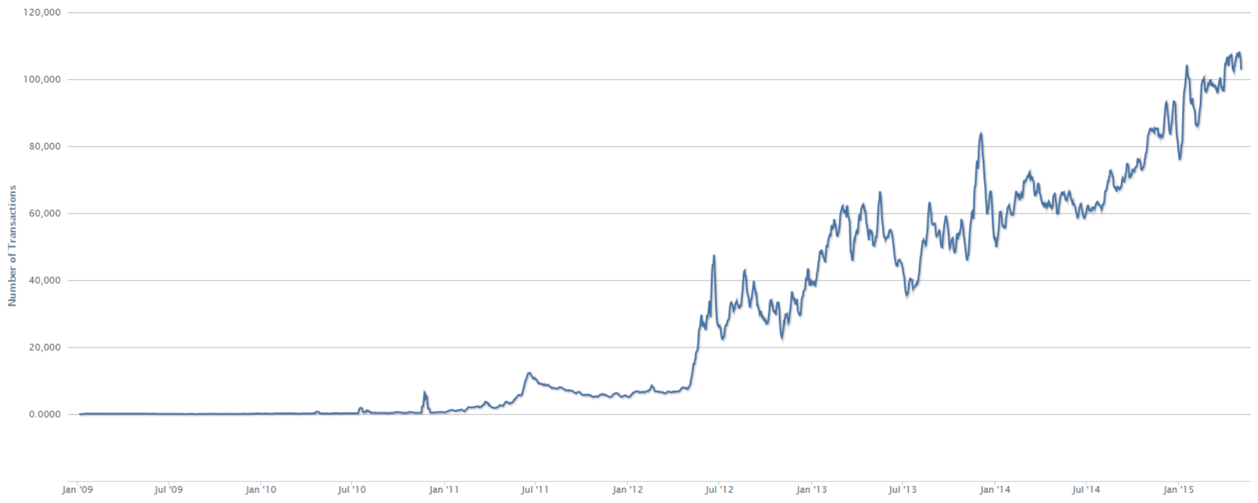


Figure 7.4: Daily transaction volume (7-day average). Source: bitcoincharts.com.

7.5: Governments Notice Bitcoin

The rest of chapter is about governments — government interaction with Bitcoin and attempts to regulate Bitcoin. Let's start with the moment when governments noticed Bitcoin, that is, when Bitcoin became big enough as a phenomenon that government started to worry about the impact it might have and how to react to it. In this section and the next we'll discuss why governments might worry about Bitcoin specifically. Then in Section 7.7 we'll turn to areas where Bitcoin businesses may be regulated for similar reasons as other other types of businesses. Finally in Section 7.8 we'll look at a case study of a proposed regulation that combines elements of regular consumer financial protection with Bitcoin-specific aspects.

Capital controls. One reason why governments would notice a digital currency like Bitcoin is that untraceable digital cash, if it exists, defeats capital controls. Capital controls are rules or laws that a country has in place that are designed to prevent the flow of value, of capital, of wealth, either in or out of the country. By putting controls on banks, investments, and so on, the country can try to prevent these flows.

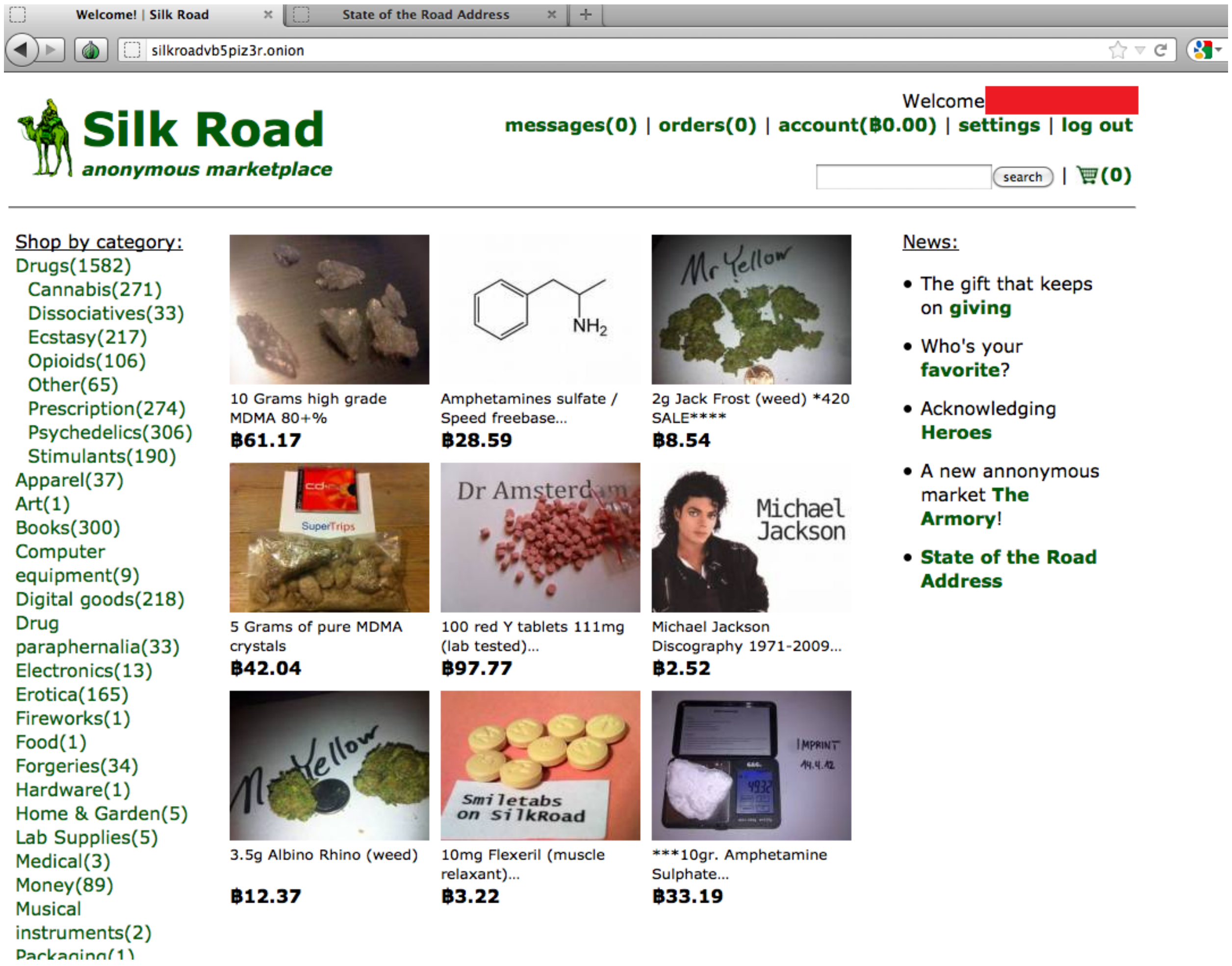
Bitcoin is a very easy way, under some circumstances, to defeat capital controls. Someone can simply buy bitcoins with capital inside the country, transmit those bitcoins outside the country electronically, and then trade them for capital or wealth outside the country. That would let them move capital or wealth from inside to outside and similarly they can move capital from outside to inside. Because wealth in this electronic form can move so easily across borders and can't really be controlled, a government that wants to enforce capital controls in a world with Bitcoin has to try to disconnect the Bitcoin world from the local fiat currency banking system. That would make it infeasible for someone to turn large amounts of local currency into Bitcoin, or large amounts of Bitcoin into local currency. We do see countries trying to beef up or protect their capital controls to do that, China being a notable example. China has engaged in increasingly strong measures to try to disconnect bitcoins from the Chinese fiat currency banking system.

Crime. Another reason governments might worry about untraceable digital cash is that it makes certain kinds of crimes easier — in particular, crimes like kidnapping and extortion that involve the payment of a ransom or payoff. Those crimes become easier when payment can be done at a distance and anonymously.

Law enforcement against kidnappers, for example, often has relied upon exploiting the hand-off of money from the victim or the victim's family to the criminals. When that can be done by email and at a distance in an anonymous way, it becomes much harder for law enforcement to follow the money. Another example: the "CryptoLocker" malware encrypts victims' files and demands ransom in Bitcoin (or other types of electronic money) to decrypt them. So the crime and the payment are both carried out at a distance. Similarly, tax evasion becomes easier when it's easier for people to move money around and to engage in transactions that are not easily tied to a particular individual or identity. Finally, the sale of illegal items becomes potentially easier when the transfer of funds can happen at a distance and without needing to go through a regulated institution.

Silk Road. A good example of that is Silk Road, which was essentially the eBay for illegal drugs. Figure 7.5 shows screenshot of Silk Road's website when it was operating. It calls itself an anonymous marketplace. Illegal drugs were the primary things for sale, with a smattering of other categories that you can see on the left. It was the largest online market for illegal drugs.

Silk Road allowed sellers to advertise goods for sale and buyers to buy them. The goods were delivered typically through the mail or through shipment services and payment was made in bitcoins. The website operated as a Tor hidden service, a concept we discussed in Chapter 6. As you can see in the screenshot, its address was `http://silkroadvb5piz3r.onion`. This way the location of the server was hidden from law enforcement. Due to the use of bitcoins for payment it was also difficult for law enforcement to follow the money and figure out who the people participating in the market were.


Welcome! | Silk Road

State of the Road Address

silkroadvb5piz3r.onion

Silk Road

anonymous marketplace

Welcome [redacted]

messages(0) | orders(0) | account(฿0.00) | settings | log out

search | **(0)**

Shop by category:

- Drugs(1582)
  - Cannabis(271)
  - Dissociatives(33)
  - Ecstasy(217)
  - Opioids(106)
  - Other(65)
  - Prescription(274)
  - Psychedelics(306)
  - Stimulants(190)
- Apparel(37)
- Art(1)
- Books(300)
- Computer equipment(9)
- Digital goods(218)
- Drug paraphernalia(33)
- Electronics(13)
- Erotica(165)
- Fireworks(1)
- Food(1)
- Forgeries(34)
- Hardware(1)
- Home & Garden(5)
- Lab Supplies(5)
- Medical(3)
- Money(89)
- Musical instruments(2)
- Packaging(1)

10 Grams high grade MDMA 80+%
฿61.17

Amphetamines sulfate / Speed freebase...
฿28.59

2g Jack Frost (weed) *420 SALE****
฿8.54

5 Grams of pure MDMA crystals
฿42.04

100 red Y tablets 111mg (lab tested)...
฿97.77

Michael Jackson Discography 1971-2009...
฿2.52

3.5g Albino Rhino (weed)
฿12.37

10mg Flexeril (muscle relaxant)...
฿3.22

***10gr. Amphetamine Sulphate...
฿33.19

News:

- The gift that keeps on **giving**
- Who's your **favorite**?
- Acknowledging **Heroes**
- A new annonymous market **The Armory**!
- **State of the Road Address**

Figure 7.5: Screenshot of Silk Road website (April 2012).

Silk Road held the bitcoins in escrow while the goods were shipped. There was an innovative escrow system which helped protect the buyers and sellers against cheating by other parties. The bitcoins would be released once the buyer certified that the goods had arrived. There was also an eBay-like reputation system that allowed buyers and sellers to get reputations for following through on their deals, and by using that reputation system Silk Road was able to give the market participants an incentive to play by the rules. So, Silk Road was innovative among criminal markets in finding ways of enforcing the rules of the criminal market at a distance, which is something that criminal markets in the past have had difficulty doing.

Silk Road was run by a person who called himself Dread Pirate Roberts — obviously a pseudonym, and you might recognize the reference. It operated from February 2011 until October 2013. Silk road was shut down after the arrest of its operator Ross Ulbricht. Ulbricht had tried to cover his tracks by operating pseudonymous accounts and by using Tor, anonymous remailers, and so on. The government was nevertheless able to connect the dots and connect him to Silk Road activity — to the servers and the bitcoins he controlled as the operator of Silk Road. He was convicted of various crimes relating to operating Silk Road. He was also charged with attempted murder for hire, although fortunately he was bad at it and nobody actually got killed.

In the course of taking down Silk Road, the FBI seized about 174,000 bitcoins, worth over $30 million at the time. As with the proceeds of any crime under US law, they could be seized by the government. Later the government auctioned off a portion of the seized bitcoins.

Lessons from Silk Road. There are several lessons from Silk Road and from the encounter between law enforcement and Ulbricht. First it's pretty hard to keep the real world and the virtual world separate. Ulbricht believed that he could live his real life in society and at the same time have a secret identity in which he operated a sizeable business and technology infrastructure. It's difficult to keep these separate worlds completely apart, and not accidentally create some linkage between them. It's hard to stay anonymous for a long time while being active and engaging in a course of coordinated conduct working with other people over time. If there's ever a connection between those two identities — say, if you slip up and use the name of one while wearing the mask of another — that link can never be destroyed and over time the different anonymous identities or mask that someone is trying to use tend to get connected.

Another lesson is that law enforcement can follow the money. Even before Ulbricht's arrest, the government knew that certain Bitcoin addresses were controlled by the operator of Silk Road, and they were watching those addresses. The result is that Ulbricht, while wealthy according to the block chain, was not actually able to benefit from that wealth because any attempt to transfer those assets over into the dollar world would have resulted in a traceable event, and probably would have resulted in rapid arrest. So although Ulbricht was the owner of something like 174,000 bitcoins, the fact is is that he was not living like a king. He lived in a one-bedroom apartment in San Francisco while apparently unable to get to the wealth that he controlled.

In short, if you intend to operate an underground criminal enterprise — and we hope you don't — then it's a lot harder to do then you might think. Technologies like Bitcoin and Tor are not panaceas for people who want to do these things and law enforcement has significant tools and resources that they can still use. Although there's been some panic in the world of law enforcement over the rise of Bitcoin, they are starting to realize that they can still follow the money up to a point and they still do have a substantial ability to investigate crimes and to make life difficult for people who want to engage in coordinated criminal action.

At the same time, we don't mean to suggest that by taking down Silk Road, law enforcement has shut down Bitcoin-based hidden markets for illegal drugs for good. In fact, after the demise of Silk Road there has been a mushrooming of such markets. Some of the more prominent ones are Sheep Marketplace, Silk Road 2, Black Market Reloaded, Evolution, and Agora. Most of these are now defunct, either due to law-enforcement actions or due to theft, often by insiders. To address the security risk of the site operator disappearing with buyers' escrowed funds, the newer marketplaces use multi-signature escrow (which we saw in Chapter 3) rather than Silk Road's model of depositing the funds with the market operator.

7.6: Anti Money Laundering

In this section we'll look at money laundering and the Anti Money Laundering (AML) rules that governments have imposed, especially in the US, that effect some Bitcoin-related businesses.

The goal of anti-money-laundering policy is to prevent large flows of money from crossing borders or moving between the underground and legitimate economy without being detected. Earlier we looked at capital controls that exist to prevent money from crossing borders. In some cases, countries are just fine with money crossing borders, but they want to know who's transferring what to whom and where that money came from.

Anti-money laundering is aimed at trying to make certain kinds of crime more difficult, especially organized crime. Organized crime groups often find themselves getting a lot of money coming in in one place and wanting to ship it somewhere else, but not wanting to explain where that money came from — hence the desire to get money across borders. Or they might find themselves making a lot of money in an underground economy and wanting to get that money into the legitimate economy so that they can spend it on sports cars and big houses or whatever it is that the leaders of the group want to do. Anti-money laundering, then, has the goals of making it harder to move money around this way and making it easier to catch people trying to do it.

Know Your Customer. One of the rules that goes with anti money laundering is something called Know Your Customer, sometimes called KYC. The details can be a bit complicated and will depend on your locale, but the basic idea is this: Know Your Customer rules require certain kinds of businesses that handle money to do three things:

1. *Identify and authenticate clients* — get some kind of authentication that clients really are who they claim they are and that those claimed identities correspond to some kind of real-world identity. So a person can't just walk in and they're john Smith from 123 Main Street in AnyTown, USA. — they have to provide an identity and have that be checked — in order to engage in certain kinds of business.
2. *Evaluate risk of client* — determine the risk of a certain client engaging in underground activities. This will be based on how the client behaves — how longstanding their business relationship is with the company, how well known they are in the community, and various other factors. KYC rules generally require covered companies to treat clients whose activities seem riskier with more attention.
3. *Watch for anomalous behavior* — that is, behavior that seems to be indicative of money laundering or criminal activity. KYC will often require a company to cut off business with a client who looks dodgy, or who is unable to authenticate themselves or their activities sufficiently for the rule.

Mandatory reporting. There are mandatory reporting requirements in the United States that are worth talking about. Companies in a broad range of sectors have to report currency transactions that are over $10,000. They must file what's called a currency transaction report to say what the transaction is and who the other party to the transaction is. There is also some requirement to authenticate who that party is. Once reported, the information goes into government databases and then might be analyzed to look for patterns of behavior that are indicative of money laundering.

Companies are also required to watch for clients who might be "structuring" transactions to avoid reporting, like engaging in a series of $9,000 transactions to get around the $10,000 reporting rule. Companies that see evidence of structuring must report it by filing a Suspicious Activity Report. Again, the information goes into a government database and might lead to investigation of the client.

The requirements here differ by country. We're not by any means trying to give you legal advice about whether you need this or what you have to do. This discussion is meant to give you an idea about what kind of requirements are imposed by anti money laundering rules. That said, take note that governments — in the U.S. and other countries— take anti money laundering rules very, very seriously. These aren't the kind of rules that you can just blow off and deal with if you get a complaint from the government later.

Bitcoin businesses have been shut down — sometimes temporarily, sometimes permanently. Business people have been arrested, and people have gone to jail for not following these rules. This is an area where government will enforce the law vigorously, regardless of whether fiat currency or Bitcoin is used. Government has enforced these laws against Bitcoin-based businesses ever since they noticed that Bitcoin was large enough to pose a risk of money laundering. If you're interested in starting any kind of business that will handle large volumes of currency, you'll need to talk to a lawyer who understands these rules.

7.7: Regulation

Now let's directly address the 'R' word — regulation. Regulation often gets a bad name, especially among the kind of people who tend to like Bitcoin. As the argument goes, regulation is some bureaucrat who doesn't know my business or what I'm trying to do, coming in and messing things up. It's a burden. It's stupid and pointless. This argument is pretty common and well understood, and it's often correct. We won't repeat it here.

Instead, in this section we'll look in some detail at reasons why regulations might sometimes be justified, because that argument is not as well understood. To be clear, the fact that we're spending most of this section talking about why regulation might be good shouldn't be read as an endorsement of widespread regulation. It's simply that we want to bring a bit more balance to the discussion in a community where regulation is often considered as always bad, or just stupid by nature.

The bottom line argument in favor or regulation is this: when markets fail and produce outcomes that are bad — and agreed to be bad by pretty much everyone in the market — then regulation can step in and try to address the failure. So the argument for regulation, when there is an argument, starts with the idea that markets don't always give you the result that you'd like.

Lemons market. Let's discuss one way in which the market can fail, a classic example called the lemons market. The name originated in the context of used cars. But let's talk about a market in concept, a market for "widgets," some kind of good that one wants to buy and sell. Let's say that widgets can either be of low quality or high quality. A high-quality widget costs a little bit more to manufacture than a low-quality widget, but it's much, much better for the consumer who buys it. Consumers like high-quality widgets a lot better.

If the market is operating well, if it's *efficient* as economists call it, it will deliver mostly high-quality widgets to consumers. That's because even though the high-quality widget is a bit costlier, most consumers prefer it and are willing to pay more for it. So under certain assumptions a market will provide this happy outcome.

On the other hand, let's suppose customers can't tell apart a low-quality widget from a high-quality widget before buying it. The classic example is the used car. You're looking at a used car sitting on the lot, and it may look pretty good, but you can't really tell if it's going to break down tomorrow or if it's going to run for a long time. The dealer probably knows if it's a lemon, but you as the customer can't tell the difference.

Let's think about the incentives that drive people in this kind of lemons market. As a consumer, you're not willing to pay extra for a high-quality widget, because you just can't tell the difference. Even if the used car dealer says that a car is perfect and is only an extra hundred dollars, you don't have a good reason to trust the dealer.

As a consequence, producers can't make any extra money by selling a high-quality widget. In fact, they lose money by selling a high-quality widget because it costs a bit more to produce and they don't get any price premium. So the market gets stuck at an equilibrium where only low-quality widgets are produced, and consumers are relatively unhappy with them.

This outcome is worse for everybody than a properly functioning market would be. It's worse for buyers because they have to make do with low-quality widgets. In a more efficient market they could have bought a widget that was much, much better for a slightly higher price. It's also worse for producers — since the widgets that are on the market are all lousy, consumers don't buy very many widgets. The widget market is relatively small, and so there's less money to be made selling widgets than there would be in a healthy market.

That's a market failure. This one, in particular, is a result of "asymmetric information" between buyers and sellers about the condition of the product. The resulting market is sometimes called a lemons market.

Fixing a lemons market. There are some market-based approaches that try to fix a lemons market. The first relies on seller reputation. The idea is that if a seller consistently tells the truth to consumers about which widgets are high vs. low quality, then the seller might acquire a reputation for telling the truth. Once they have that reputation, they may be able to sell high-quality widgets for a higher price because consumers will believe them, and therefore the market can operate more efficiently.

This sometimes works and sometimes doesn't depending on the precise assumptions you make about the market. Of course, it will never work as well as a market where consumers can actually tell the difference in quality. For one thing, it takes a while for a producer to build up a good reputation. That means they have to sell high-quality widgets at low prices for a while until consumers learn that they’re telling the truth. That makes it harder for an honest seller to get into the market.

The other potential problem is that a seller, even if they've been honest up to now, no longer has the incentive to be honest if they want to get out of the market (say, if their sales are shrinking). In that case their incentive is to massively cheat people all at once and then exit the market. So reputation doesn’t work well at the beginning of a seller’s presence is in the market as well as at the end.

A reputation-based approach also tends not to work in businesses where consumers don't do repeat business with the same entity, or where the product category is very new, and therefore there hasn’t been enough time for sellers to build up a reputation. A high-tech market like Bitcoin exchanges suffers just those problems.

The other market-based approach is warranties. The idea is that a seller could provide a warranty to a buyer that says if the widget turns out to be low quality, the seller will provide an exchange or a refund. That can work well up to a point, but there's also a problem: a warranty is just another kind of product that can also come in high-quality or low-quality versions! A low-quality warranty is one where the seller doesn't really come through when you come back with the broken product. They renege on their promise or they make you jump through all kinds of hoops.

Regulatory fixes. So if a lemons market has developed, and if these market-based approaches don't work for the particular market, then regulation might be able to help. Specifically, there are three ways in which regulation might be able to address the problem.

First, regulation could require disclosure. It could require, say, that all widgets be labeled as high quality or low quality, combined with penalties on the firms for lying. That gives consumers the information that they were missing. A second approach to regulation is to have quality standards so that no widget can be sold unless it meets some standard of quality testing, and to have that standard set so that only high-quality widgets can pass the test. That would result in a market that again has only one kind of widget, but at least it's high-quality widgets, assuming that the regulation works as intended. The third approach is to require all sellers to issue warranties and then enforce the operation of those warranties so that sellers are held to the promises that they make.

Any of these forms of regulation could obviously fail — it might not work as intended, might be mis-written or misapplied, or might might be burdensome on sellers. But there's at least the possibility that regulation of this type might help to address the market failure due to a lemons market. People who argue for regulation of Bitcoin exchanges, for example, sometimes point to them as an example of a lemons market.

Collusion and antitrust law. Another example of markets not operating the way we would like them to is price fixing. Price fixing is when different sellers collude with each other and agree to raise prices or to not lower them. A related situation is where companies that would otherwise go into competition with each other agree not to compete. For example, if there were two bakeries in town they might agree that one of them will only sell muffins and the other will only sell bagels, and that way there's less competition between them then there would be if they both sold muffins and bagels. As a result of the reduced competition presumably prices go up, and the merchants are able to foil the operation of the market.

After all, the reason that the market protects consumers well in its normal operation is through the vehicle of competition. Sellers have to compete in order to offer the best goods at the best price to consumers, and if they don't compete in that way then they won't get business. An agreement to fix prices or to not compete circumvents that competition. When people take steps that prevent competition, that's another kind of market failure.

These kinds of agreements — to raise prices or to not compete — are illegal in most jurisdictions. This is part of antitrust law or competition law. The goal of this body of law is to prevent deliberate actions that prevent or harm competition. More generally, it limits actions other than simply offering good products at good prices, such as attempts to reduce competition through mergers. Antitrust law is very complicated and we've given you only a sketch of it, but it's another instance of how the market can fail and how the law can and will step in to prevent it.

7.8: New York's BitLicense Proposal

So far we've discussed regulation in general: different forms of regulation, why regulation might be justified in some cases and might make good economic sense. Now let's turn to a specific effort by a specific state to introduce specific regulation of Bitcoin, namely New York State's BitLicense proposal. The information here is current as of early 2015, but the landscape of Bitcoin regulation changes quickly. That doesn't matter much for our purposes, because our goal isn't so much to help you understand a specific piece of actual or proposed regulation. Rather, we want to help you understand the kinds of things regulators are doing and give you a sense of how they think about the problem.

The BitLicense proposal was issued in July 2014 and has since been revised in response to comments from the Bitcoin community, industry, the public, and other stakeholders. It was issued by the New York State Department of Financial Services, the part of the state of New York that regulates the financial industry. Of course, the state of New York has the world’s largest financial center, and so it's a part of the State government that is use to dealing with relatively large institutions.

Who’s covered. BitLicense is a proposed set of codes, rules, and regulations that has to do with virtual currencies. Fundamentally, it says that you’d need to get something called a BitLicense from the New York Department of Financial Services if you wanted to do any of the things listed in the text below.

> Virtual Currency Business Activity means the conduct of any one of the following types of activities involving New York or a New York Resident:
>
> 1. receiving Virtual Currency for Transmission or Transmitting Virtual Currency, except where the transaction is undertaken for non-financial purposes and does not involve the transfer of more than a nominal amount of Virtual Currency;
> 2. storing, holding, or maintaining custody or control of Virtual Currency on behalf of others;
> 3. buying and selling Virtual Currency as a customer business;
> 4. performing Exchange Services as a customer business; or
> 5. controlling, administering, or issuing a Virtual Currency.
>
> The development and dissemination of software in and of itself does not constitute Virtual Currency Business Activity.

The text refers to “activities involving New York or a New York Resident,” reflecting the regulatory authority of NYDFS. Yet the impacts of regulations like these extend well beyond the borders of the state, for two reasons. First, for states with significant populations such as New York or California, faced with the choice between complying with state laws and not doing business with consumers in those states, most companies will choose to comply. Second, some states are generally perceived as leaders in regulating certain economic sectors — finance in the case of New York, technology in the case of California. That means that other U.S. states often follow the direction that they set.

Notice the exception for non-financial uses in the first category — this was added in the second revision, and it is a good one. It’s a carve-out for just the kind of Bitcoin-as-a-platform applications that we’ll look at starting in Chapter 9. The second category might cover things like wallet services. As for the third category, it appears that you can buy and sell bitcoins for yourself, but doing it as a customer business requires a BitLicense. The fourth category is self-explanatory. The final one might apply more to altcoins, many of which are somewhat centralized, than to Bitcoin. We’ll look at altcoins in Chapter 10.

The software-development exception at the end is again an important one. The language wasn’t in the original version, and there was an outcry from the community. NYDFS superintendent Benjamin Lawsky clarified soon after that the intent was not to regulate developers, miners, or individuals using Bitcoin. The second version contains the explicit language above.

Requirements. If the regulation goes into effect and you’re one of the covered entities, you’ll have to apply for a license. To apply for a license there's detailed language in the proposal which you can read, but roughly speaking you have to provide information on the ownership of your enterprise, on your finances, and insurance, on your business plan — generally to allow the NYDFS to know who you are, how well-backed you are, where your money comes from, and what you're planning to do. And you have to pay an application fee.

Once you had a license, you'd have to provide updated information to NYDFS about the things we listed: ownership, finances, insurance, and so on. You'd have to provide periodic financial statements so they could keep track of how you're doing financially. You'd be required to maintain a financial reserve, the amount of which will be set by NYDFS based on various factors about your business.

There are detailed rules about things like how you would keep custody of consumer assets. There are anti-money laundering rules which might or might not go beyond what's already required by existing laws. There are rules about having a security plan and penetration testing and so on. There are rules about disaster recovery — you have to have a disaster-recovery plan that meets various criteria. There are rules about record keeping — you have to keep records, and make them available to the NYDFS under certain circumstances. You have to have written policies about compliance and you have to designate a compliance officer —someone within your organization who's in charge of compliance and has the necessary responsibility and authority. There's a requirement that you disclose risk to consumers, so that consumers understand the risks of doing business with you.

As you can see, the requirements are substantial, and they're analogous to the sort of requirements for a mutual fund or a publicly traded stock. The NYDFS must still decide what to do with the proposal — whether to withdraw it, issue it in its current form, or make further modifications. Along with that decision they'll issue some kind of a document that gives the rationale for what they decided to do.

If something like the BitLicense goes into effect, it would really be a major step in the history of Bitcoin. You would have a situation where not only NYDFS, but perhaps other jurisdictions would start to step in and regulate, and you'd start to see Bitcoin businesses start to get closer to the traditional model of regulated financial institutions.

This would be a step that's in some ways contrary to the cypherpunk or cypher-libertarian ideas of about what Bitcoin was suppose to be, but on the other hand there's a certain inevitability that as soon as Bitcoin became really valuable, Bitcoin businesses became big businesses, and government got interested, regulation would ensue. Bitcoin businesses touch real people and the fiat currency economy. If Bitcoin is big enough to matter, then it is big enough to get regulated. It represents a retreat from what the original advocates of Bitcoin had in mind, but in another way it represents the Bitcoin ecosystem growing up and integrating into the regular economy which is much more regulated. Regardless of your stance on it, regulation is starting to happen, and if you're interested in starting a Bitcoin business you need to be paying attention to this trend.

Will this be a success? There are different ways to look at it, but one way to evaluate the effectiveness of regulation like BitLicense from a public policy standpoint at improving the quality of Bitcoin businesses is this: if something like BitLicense goes into effect, and if companies start advertising to customers outside New York that they can be trusted because they have a BitLicense, and if that argument is convincing to consumers when they're picking a company to do business with, then regulation will be working in the way that its advocates wanted it to. Whether that will happen and how it will affect the future of Bitcoin is something that we'll have to wait and see.

Further reading

A paper that contains many interesting details of how Silk Road operated and what was sold there:

Christin, Nicolas. Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace. Proceedings of the 22nd international conference on World Wide Web 2013.

A guide to the regulatory issues that Bitcoin raises:

Brito, Jerry, and Andrea Castillo. Bitcoin: A primer for policymakers. Mercatus Center at George Mason University, 2013.

A book that looks at the history of modern cryptography and the cypherpunk movement, which gives some intuition for the early political roots of Bitcoin:

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

A popular exposition of early work on digital cash, combined with a vision for a world with digital privacy:

Chaum, David. Security without identification: Transaction systems to make big brother obsolete. Communications of the ACM, 1985.

The text of the BitLicense proposal:

New York State Department of Financial Services Regulations of the Superintendent of Financial Services. Part 200: Virtual Currencies (revised), 2015.