Bitcoin and Cryptocurrency Technologies

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

Draft — Jan 9, 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 be 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 there aren't double-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 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.

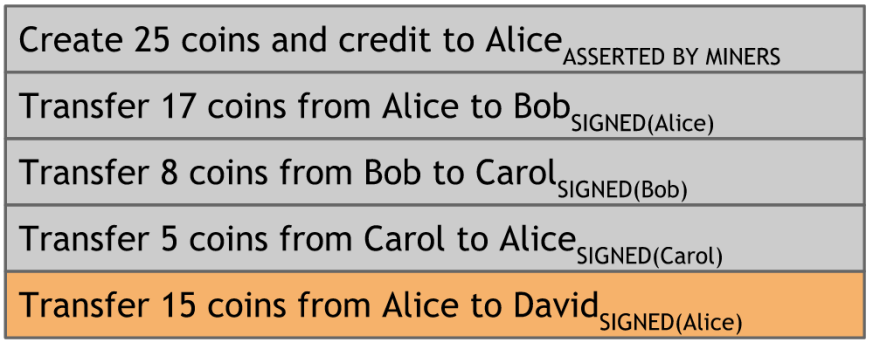


Figure 3.1 an account-based ledger

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.

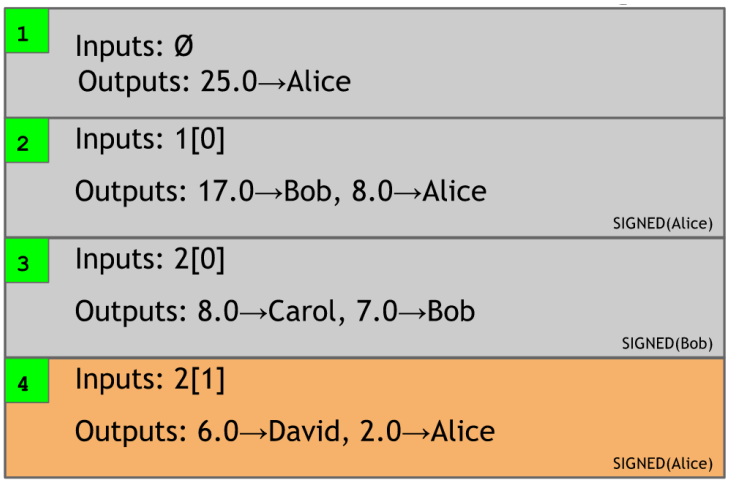


Figure 3.2 a transaction-based ledger, which is very close to Bitcoin

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 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 two 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.

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. You may be wondering why the output value would ever be less than the input value. We'll see that in Chapter 4.

  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 a set of commands. And, in fact, the 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 resulting script has to run successfully in order to claim the coins. 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.

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

Figure 3.5. To check if a transaction correctly redeems an output, we concatenate the input and output transactions. Shown here is a concatenated *scriptPubKey* and *scriptSig.* 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.

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 supported cryptographic operations natively. 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.

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 for called CHECKMULTISIG that lets you check multiple signatures with one instruction. Figure 3.5 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 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 3.5 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 3.6, 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.

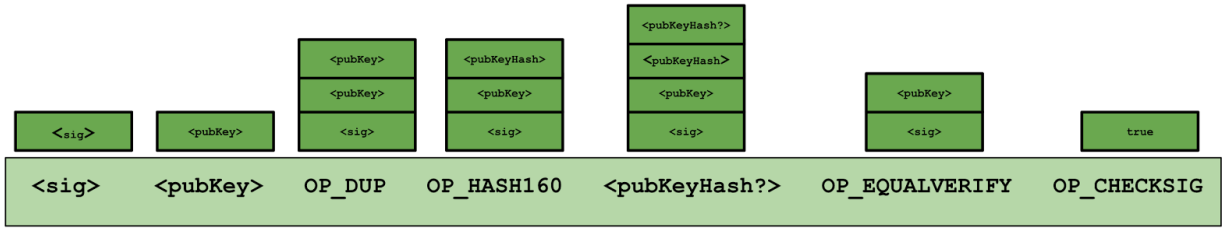


Figure 3.6 **Execution of a Bitcoin script.** On the bottom, we show the instruction in the script. Data instructions are denoted with surrounding angle brackets, whereas opcodes begin with "OP_". On the top, we show the stack just after that instruction has been executed.

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 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, a 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 quote small 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 used quite heavily 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 signature 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 go 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 signature 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, in a change address is going back to the bank, maybe. But essentially, the bank is paying Bob here from a bank-controlled address, which we call green addresses. 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 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 still 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 amount of money 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. You can cut off the service. I'm not going to pay you anymore." And Bob says, "Great. I'll disconnect your service, and I'll 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 be out 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 that in her hand. 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 smart contracts people would like to have but aren't supported by the Bitcoin scripting language today or nobody has come up with a way to do it yet. Even so, as we saw, you can do quite a bit 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 an efficient way. As we saw in Chapter 1, to prove that that 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.

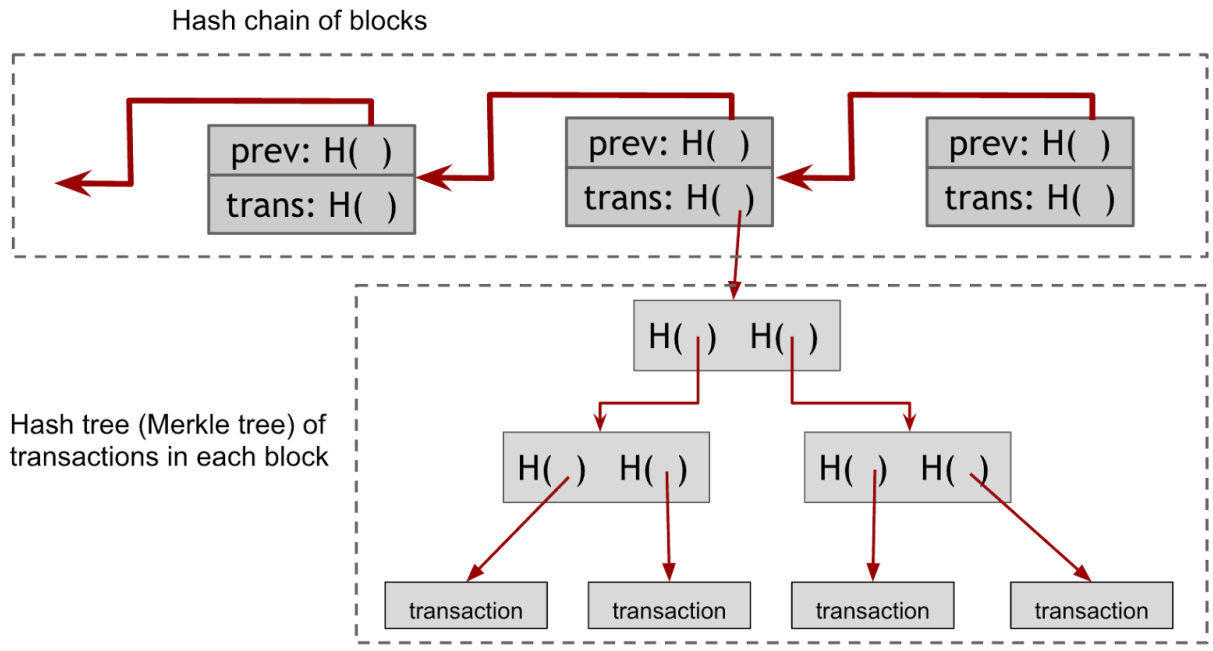


Figure 3.7. The Bitcoin block chain contains two different hash structures. The first is a hash chain of blocks that links the different blocks to one another. The second is internal to each block and is a Merkle Tree of transactions within the blocks.

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 3.8. **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 halving every 210,000 blocks (about 4 years), and the transactions 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 up 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 them in a pool of transactions that they've heard about but 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.

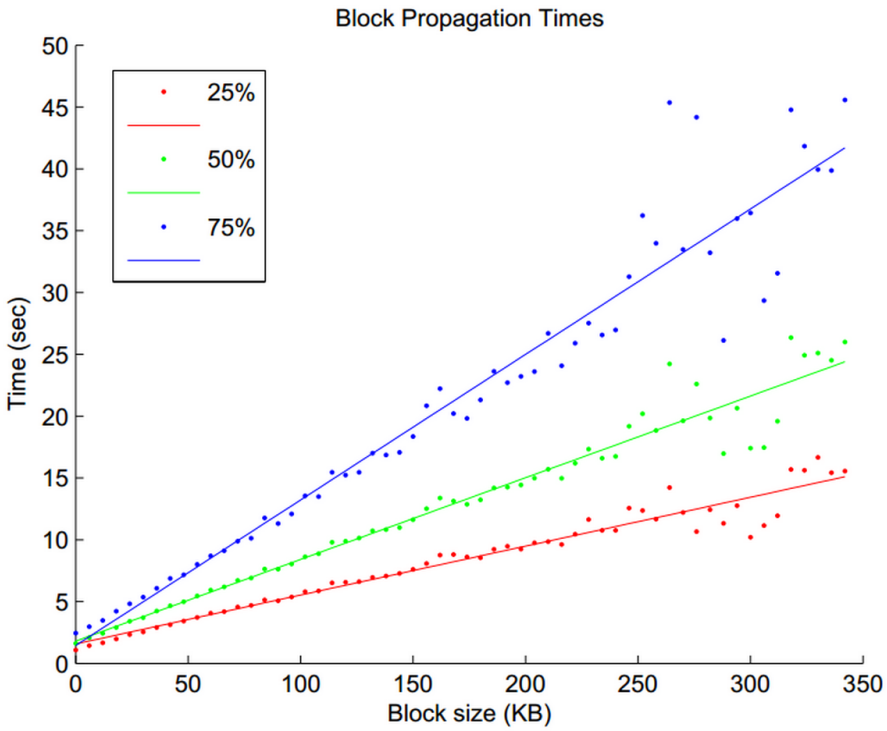
The good news is that this is perfectly okay. Whoever mines the next block will essentially break the tie and decide which of those two pending transactions should end up being put permanently into a block. Let's say the Alice → Charlie transaction makes it into the block. When nodes with the Alice → 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 flood in opposite directions, and which transaction a node sees first will depend on which side of the network they started out closer to.

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 3.9 **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 3.9 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 3.10), well within the abilities of a single commodity desktop machine.

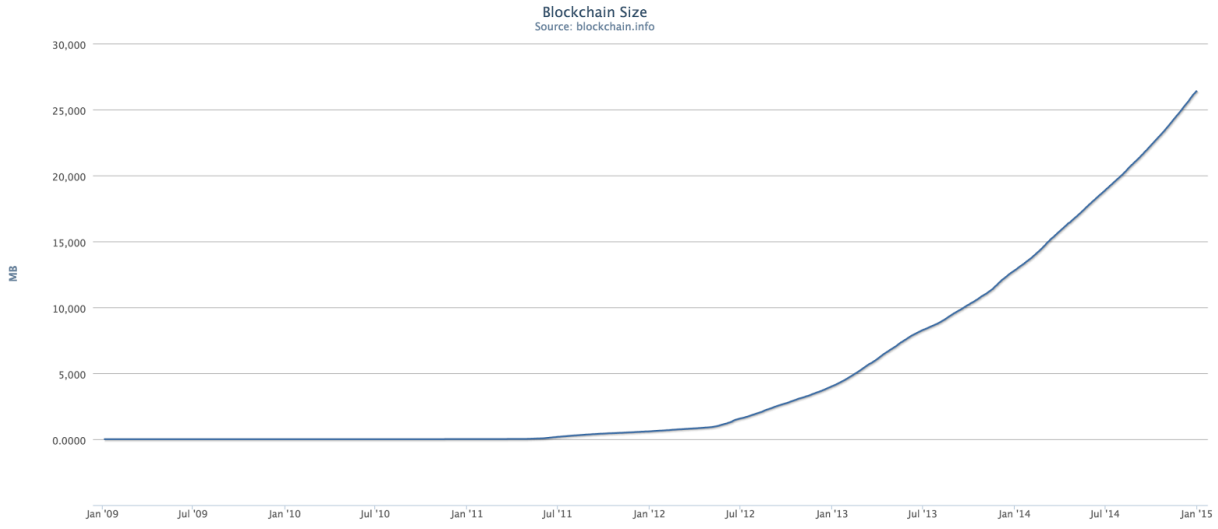


Figure 3.10. **Size of the block chain.** Fully validating nodes must store the entire block chain, which as of the end of 2014 is over 26 gigabytes.

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 when 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.

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 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 upgrades 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 additional step of verifying that value itself runs 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 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. blockchain.info is one such useful online resource.

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 making the 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?