

# MODULE 3

## 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	ASSERTED BY MINERS
Transfer 17 coins from Alice to Bob	SIGNED(Alice)
Transfer 8 coins from Bob to Carol	SIGNED(Bob)
Transfer 5 coins from Carol to Alice	SIGNED(Carol)
Transfer 15 coins from Alice to David	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.

1	Inputs: $\emptyset$ Outputs: 25.0→Alice	
2	Inputs: 1[0] Outputs: 17.0→Bob, 8.0→Alice	SIGNED(Alice)
3	Inputs: 2[0] Outputs: 8.0→Carol, 9.0→Bob	SIGNED(Bob)
4	Inputs: 2[1] Outputs: 6.0→David, 2.0→Alice	SIGNED(Alice)

**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.



**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 entities 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.

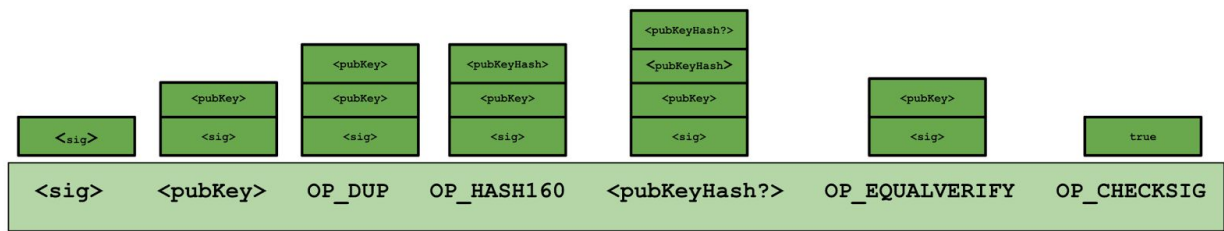
<b>OP_DUP</b>	Duplicates the top item on the stack
<b>OP_HASH160</b>	Hashes twice: first using SHA-256 and then RIPEMD-160
<b>OP_EQUALVERIFY</b>	Returns true if the inputs are equal. Returns false and marks the transaction as invalid if they are unequal
<b>OP_CHECKSIG</b>	Checks that the input signature is a valid signature using the input public key for the hash of the current transaction
<b>OP_CHECKMULTISIG</b>	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 verify that signature — specified in the scriptSig component of a transaction input in the redeeming transaction. As we mentioned, when we see a data instruction, we just push it onto the stack. The rest of the script was specified in the scriptPubKey component of a transaction output in the referenced transaction.

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 referenced transaction. It is the hash of a public key that the sender specified; the corresponding private key must 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.

We’ve already checked that this public key is in fact the public key that the referenced transaction specified, 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 are 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 a signature of? What is 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."

Bitcoin has a clever solution to this problem, and it applies to not just multi-sig addresses but to any complicated condition governing when coins can be spent. Instead of telling the sender "send your coins to the hash of this public key", the receiver can instead tell the sender "send your coins to the hash of this *script*. Impose the condition that to redeem those coins, it is necessary to reveal the script that has the given hash, and further, provide data that will make the script evaluate to true." The sender achieves this by using the Pay-to-script-hash (P2SH) transaction type, which has the above semantics.

Specifically, the P2SH script simply hashes the top value on the stack, checks if it matches the provided hash value, then executes a special second step of validation: that top data value from the stack is reinterpreted as a sequence of instructions, and executed a second time as a script, with the rest of the stack as input.

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.

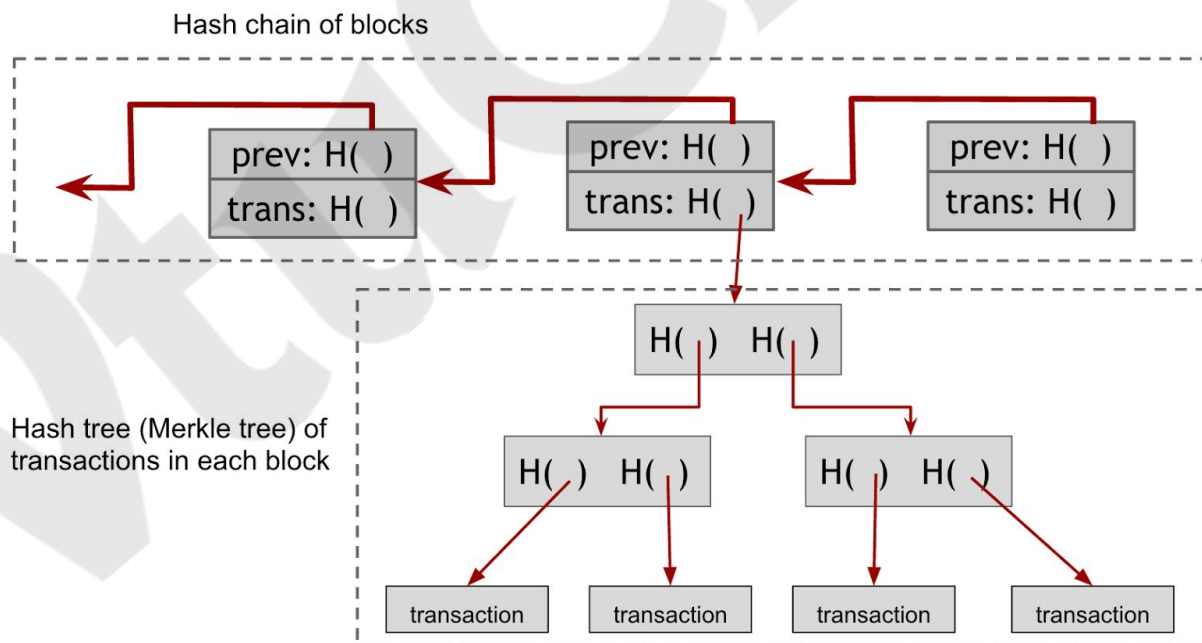
Research into smart contracts goes far beyond the applications that we saw in this section. There are many types of smart contracts which people would like to be able to enforce but which aren't

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](https://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 keep and whether or not to forward a transaction. We'll look more closely at miner incentives in Chapter 5.

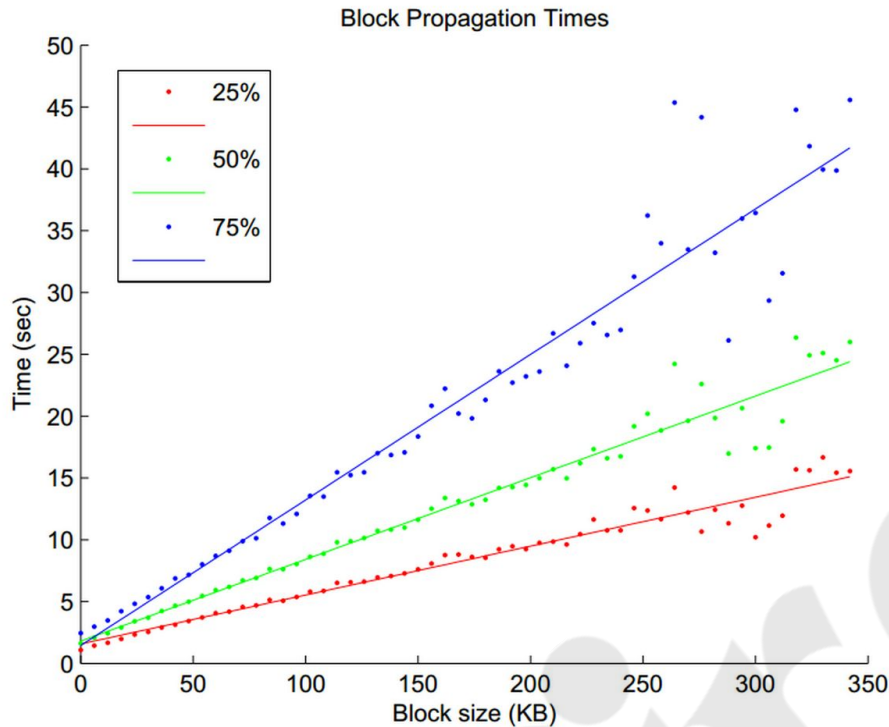
**Sidebar: Zero-confirmation transactions and replace-by-fee.** In Chapter 2 we looked at zero-confirmation transactions, where the recipient accepts the transaction as soon as it is broadcast on the network. This isn't designed to be secure against double spends. But as we saw, the default behavior for miners in the case of conflicting transactions is to include the transaction they received first, and this makes double-spending against zero-confirmation transactions moderately hard. As a result, and due to their convenience, zero-confirmation transactions have become common.

Since 2013, there has been interest in changing the default policy to *replace-by-fee* (RBF) whereby nodes will replace a pending transaction in their pool if they hear a conflicting transaction which includes a higher fee. This is the rational behavior for miners, at least in a short-term sense, as it gives them a better fee. However, replace-by-fee would make double-spending against zero-confirmation attacks far easier in practice.

Replace-by-fee has therefore attracted controversy, both in terms of the technical question of whether it is possible to prevent or deter double-spending in an RBF world, and the philosophical question of whether Bitcoin should try to support zero-confirmation as best it can, or abandon it. We won't dive into the long-running controversy here, but Bitcoin has recently adopted "opt-in" RBF whereby transactions can mark themselves (using the sequence-number field) as eligible for replacement by higher-fee transactions.

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 also 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.

**Sidebar: Bitcoin's block size conundrum.** Due to Bitcoin's growing popularity, as of early 2016 it has become common for the 1-megabyte space in blocks to be filled up within the period between blocks (especially when, due to random chance, a block takes longer than 10 minutes to find) first, resulting in some transactions having to wait one or more additional blocks to make their way into the block chain. Increasing the block size limit will require a hard fork.

The question of whether and how to address the block chain's limited bandwidth for transactions has gripped the Bitcoin community. The discussion started years ago, but with little progress toward a consensus, it has gradually gotten more acrimonious, escalating into a circus. We'll discuss Bitcoin's community, politics, and governance in Chapter 7.

Depending on the resolution of the block-size problem, some of the details in this chapter might become slightly out of date. The technical details of increasing Bitcoin's transaction-processing capacity are interesting, and we encourage you to read more online.

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.

## 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 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 your 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 digit 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.

**Vanity addresses.** Some individuals or merchants like to have an address that starts with some human-meaningful text. For example, the gambling website Satoshi Bones has users send money to addresses containing the string “bones” in positions 2--6, such as

1bonesEeTcABPjLzAb1VkFgySY6Zqu3sX (all regular addresses begin with the character 1, indicating pay-to-pubkey-hash.)

We said that addresses are outputs of a hash function, which produces random-looking data, so how did the string “bones” get in there? If Satoshi Bones were simply making up these addresses, lacking the ability to invert hash function, they wouldn’t know the corresponding private keys and hence wouldn’t actually control those addresses. Instead, they repeatedly generated private keys until they got lucky and found one which hashed to this pattern. Such addresses are called *vanity addresses* and there are tools to generate them.

How much work does this take? Since there are 58 possibilities for every character, if you want to find an address which starts with a specific  $k$ -character string, you’ll need to generate  $58^k$  addresses on average until you get lucky. So finding an address starting with “bones” would have required generating over 600 million addresses! This can be done on a normal laptop nowadays. But it gets exponentially harder with each extra character. Finding a 15-character prefix would require an infeasible amount of computation and (without finding a break in the underlying hash function) should be impossible.

**Sidebar: Speeding up vanity address generation.** In Bitcoin, if we call the private key  $x$ , the public key is  $g^x$ . The exponentiation represents what’s called scalar multiplication in an elliptic curve group. The address is  $H(g^x)$ , the hash of the public key. We won’t get into the details here, but exponentiation is the slow step in address generation.

The naive way to generate vanity addresses would be to pick a pseudorandom  $x$ , compute  $H(g^x)$ , and repeat if that address doesn’t work. A much faster approach is to try  $x+1$  if the first  $x$  fails, and continue incrementing instead of picking a fresh  $x$  each time. That’s because  $g^{x+1} = x g^x$ , and we’ve already computed  $g^x$ , so we only need a multiplication operation for each address instead of exponentiation, and that’s much faster. In fact, it speeds up vanity address generation by over two orders of magnitude.

## 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 to 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.

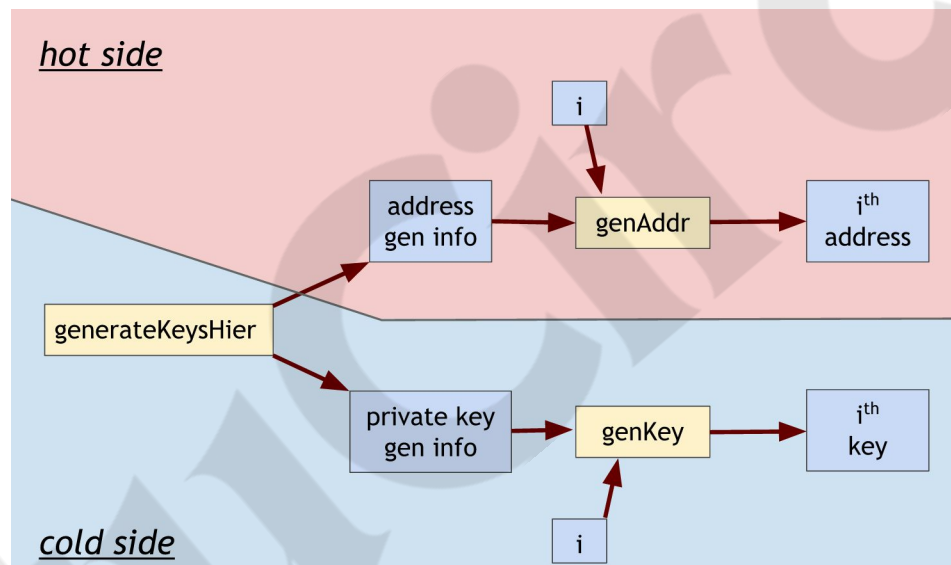
**Hierarchical wallets.** A more effective solution is to use a hierarchical wallet. It allows the cold side to use an essentially unbounded number of addresses and the hot side to know about these addresses, but with only a short, one-time communication between the two sides. But it requires a little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

Here's how it works. Recall that normally an ECDSA private key is a random number  $x$  and the corresponding public key is  $g^x$ . For hierarchical key generation, we'll need two other random values  $k$  and  $y$ .

Private key generation info:	$k, x, y$
$i^{\text{th}}$ private key:	$x_i = y + H(k \parallel i)$
Address generation info:	$k, g^y$
$i^{\text{th}}$ public key:	$g^{x_i} = g^{H(k \parallel i)} \cdot g^y$
$i^{\text{th}}$ address:	$H(g^{x_i})$

This has all the properties that we want: each side is able to generate its sequence of keys, and the corresponding keys match up because (because the public key corresponding to a private key  $x$  is  $g^x$ ). It has one other property that we haven't talked about: when you give out the public keys, those keys won't be linkable to each other, that is, it won't be possible to infer that they come from the same wallet. The straw-man solution of having the cold side generate a big batch of addresses does have this property, but we had to take care to preserve it when with the new technique considering that the keys aren't in fact independently generated. This property is important for privacy and anonymity, which will be the topic of Chapter 6.

Here we have two levels of security, with the hot side being at a lower level. If the hot side is compromised, the unlinkability property that we just discussed will be lost, but the private keys (and the bitcoins) are still safe. In general, this scheme supports arbitrarily many security levels --- hence "hierarchical" --- although we haven't seen the details. This can be useful, for instance, when there are multiple levels of delegation within a company.

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, 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.

**Sidebar: generating memorable passphrases.** One passphrase-generation procedure that gives about 80 bits of entropy is to pick a random sequence of 6 words from among the 10,000 most common English words ( $6 \times \log_2(10000)$  is roughly 80). Many people find these easier to memorize than a random string of characters. Here are a couple of passphrases generated this way.

worn till alloy focusing okay reducing  
earth dutch fake tired dot occasions

In practice, it is also wise to use a deliberately slow function to derive the private key from the passphrase to ensure it takes as long as possible for the attacker to try all possibilities. This is known as *key stretching*. To create a deliberately slow key-derivation function, we can take a fast cryptographic hash function like SHA-256 and compute say  $2^{20}$  iterations of it, multiplying the attacker's workload by a factor of  $2^{20}$ . Of course, if we make it 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.



**Figure 4.3: A Bitcoin paper wallet** with the public key encoded both as a 2D barcode and in base 58 notation. Observe that the private key is behind a tamper-evident seal.

**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*.

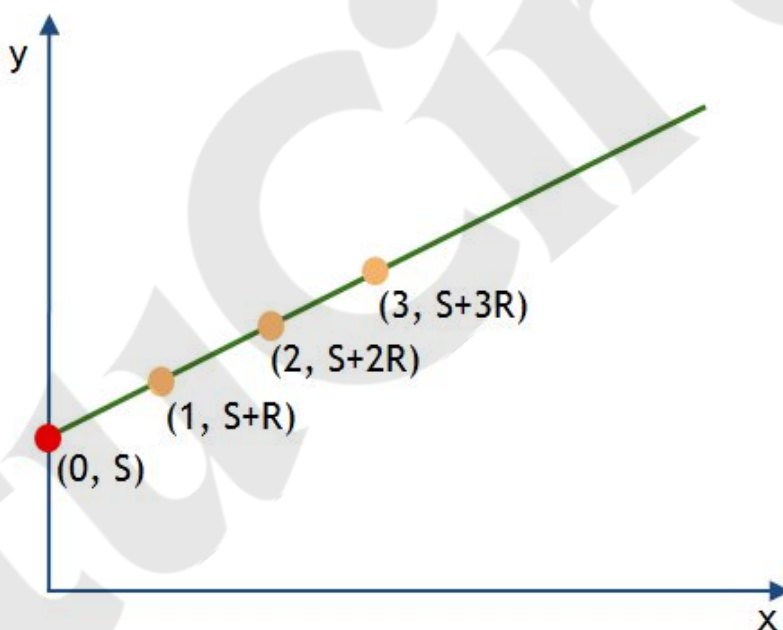
Here's the idea: we want to divide our secret key into some number  $N$  of pieces. We want to do it in such a way that if we're given any  $K$  of those pieces then we'll be able to reconstruct the original secret, but if we're given fewer than  $K$  pieces then we won't be able to learn anything about 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  $S \oplus R$ . ( $\oplus$  represents bitwise XOR). Essentially, we’ve “encrypted”  $S$  with a one-time pad, and we store the key ( $R$ ) and the ciphertext ( $S \oplus R$ ) 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 4.4: 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 4.4. 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 of 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

$$x=1, y=(S+R) \bmod P$$

$$x=2, y=(S+2R) \bmod P$$

$$x=3, y=(S+3R) \bmod P$$

and so on. The secret corresponds to the point  $x=0, y=(S+0\cdot R) \bmod P$ , 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.

In our presentation above, we motivated threshold signatures by explaining how it can help achieve two-factor (or multi-factor) security, and multi-signatures by explaining how it can help a set of individuals share control over jointly held funds. But either technology is applicable to either situation.

## 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](http://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 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 one account to another. All that happens in this transaction is that the exchange is now making a different promise to me than 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 they 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

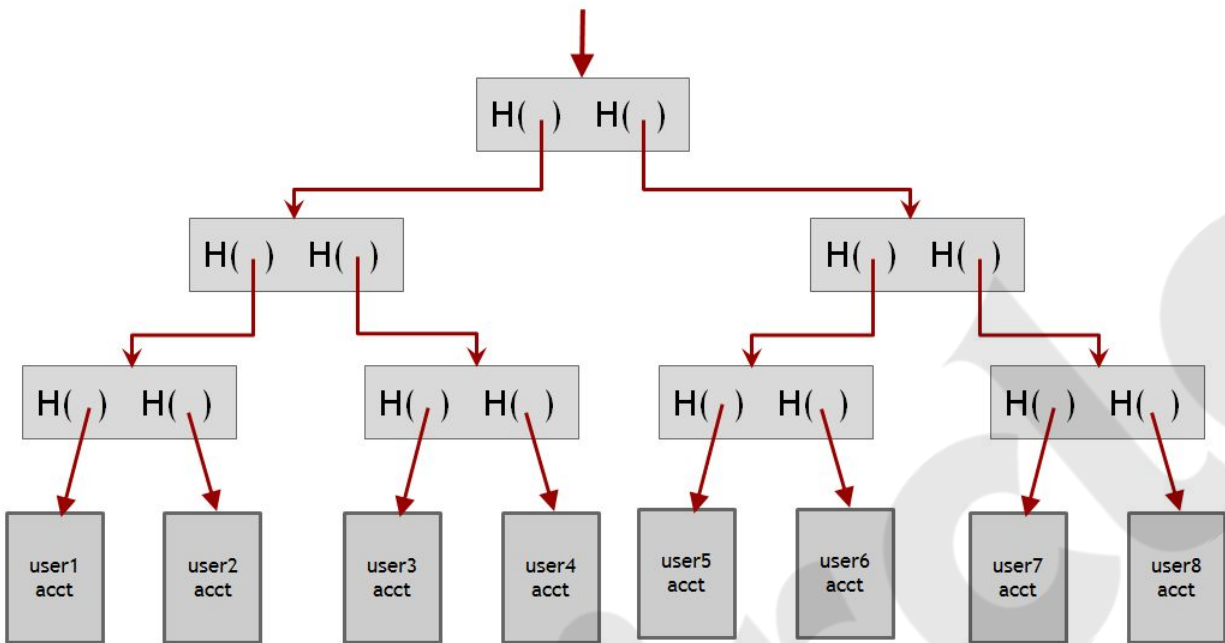
the same private key that was used to sign the payment-to-self transaction. This proves that someone who knew that private key participated in the proof of reserve.

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 is 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 your 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 4.5: 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.

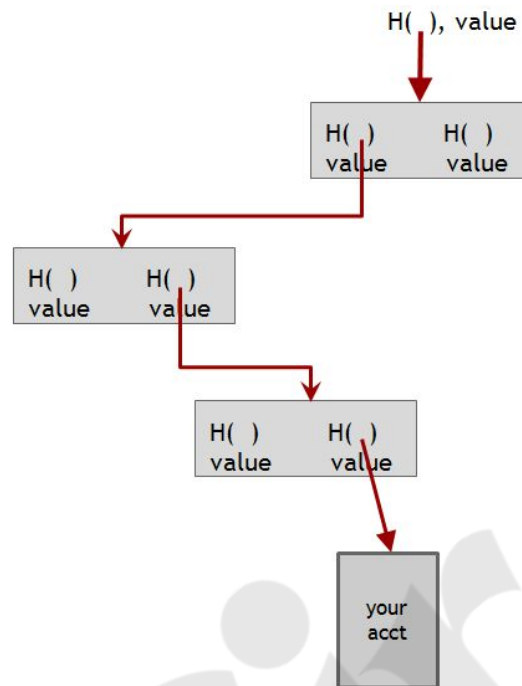
Now, we're going to add to each one of these hash pointers another field, or attribute. This attribute is a number that represents the total monetary value in bitcoins of all deposits that are in the sub-tree underneath that hash pointer in the tree. For this to be true, the value corresponding to each 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 4.6. 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.

- Neither of the values is a negative number.



**Figure 4.6: 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.

You might notice that the two proofs presented here (the proof of reserves by signing a challenge string and the proof of liabilities via a Merkle tree) reveal a lot of private information. Specifically, they reveal all of the addresses being used by the exchange, the total value of the reserves and liabilities, and even some information about the individual customers balances. Real exchanges are hesitant to publish this, and as a result cryptographic proofs of reserve have been rare.

A recently proposed protocol called Provisions enables the same proof-of-solvency, but without revealing the total liabilities or reserves or the addresses in use. This protocol uses more advanced

crypto and we won't cover it here, but it's another example showing how cryptography can be used to ensure privacy.

Solvency 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.

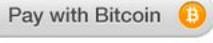
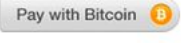


The merchant also wants low risk. There are various possible risks: using new technology may cause their website to go down, costing them money. There's the security risk of handling bitcoins — someone might break into their hot wallet or some employee will make off with their bitcoins. Finally there's 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 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 ☒  ☐   
☐  ☐ 

Item Name  Amount

Item Description

Send Funds To

[Show Advanced Options](#)

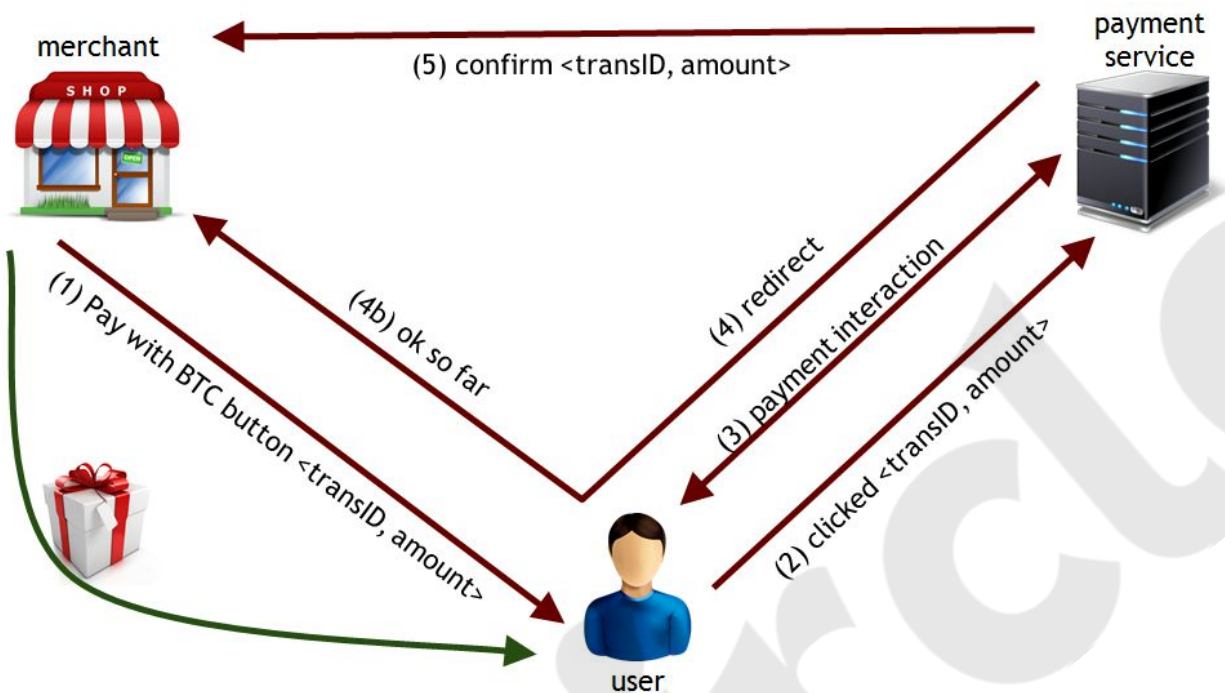
**Figure 4.7: Example payment service interface for generating a pay-with-Bitcoin button.** A merchant can use this interface to generate a HTML snippet to embed on their website.

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 and fills out a form describing the item, price, and presentation of the payment widget, and so on. Figure 4.7 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.





**Figure 4.8: 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 4.8.

1. The user picks out an item to buy on the merchant website, and when it comes time 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 saying that the 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 it 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 complex, but we'll limit ourselves to 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 are 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:  $(\text{sum of input age} * \text{input value}) / (\text{transaction size})$ . 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 2015 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 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 its 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](http://bitcoincharts.com) that show 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](http://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. As of October 2015 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 include demand deposits of bitcoins. That is, if someone has put money into their account in a Bitcoin exchange, and the exchange doesn't maintain a full reserve to meet every single deposit, then there will be demand deposits at that exchange that are larger than the number of coins that the exchange is holding.

Depending on what question you're asking about the market it might or might not be correct to include demand deposits in the supply. Basically, you should include demand deposits in a market analysis when demand-deposited money can be sold in that market. For example, if you've traded dollars for a demand deposit of bitcoins, and the exchange allows demand-deposited bitcoins to be redeemed 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 13.9 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 a 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 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 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 the purpose of mediating transactions. 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 the 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.

Notice 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.

VitalCircles