CHAPTER 6

Transactions

Introduction

Transactions are the most important part of the bitcoin system. Everything else in bitcoin is designed to ensure that transactions can be created, propagated on the network, validated, and finally added to the global ledger of transactions (the blockchain). Transactions are data structures that encode the transfer of value between participants in the bitcoin system. Each transaction is a public entry in bitcoin's blockchain, the global double-entry bookkeeping ledger.

In this chapter we will examine all the various forms of transactions, what they contain, how to create them, how they are verified, and how they become part of the permanent record of all transactions. When we use the term "wallet" in this chapter, we are referring to the software that constructs transactions, not just the database of keys.

Transactions in Detail

In Chapter 2, we looked at the transaction Alice used to pay for coffee at Bob's coffee shop using a block explorer (Figure 6-1).

The block explorer application shows a transaction from Alice's "address" to Bob's "address." This is a much simplified view of what is contained in a transaction. In fact, as we will see in this chapter, much of the information shown is constructed by the block explorer and is not actually in the transaction.

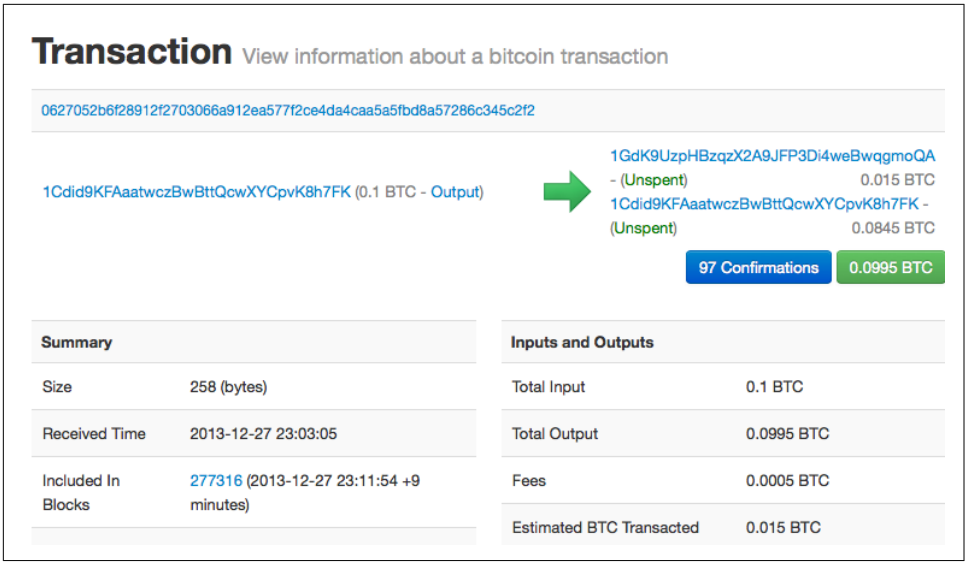

Figure 6-1. Alice's transaction to Bob's Cafe

Transactions—Behind the Scenes

Behind the scenes, an actual transaction looks very different from a transaction provided by a typical block explorer. In fact, most of the high-level constructs we see in the various bitcoin application user interfaces *do not actually exist* in the bitcoin system.

We can use Bitcoin Core's command-line interface (`getrawtransaction` and `decoderawtransaction`) to retrieve Alice's "raw" transaction, decode it, and see what it contains. The result looks like this:

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid":
"7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204
b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d1
72787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",
      "sequence": 4294967295
    }
  ],
  "vout": [
```

```
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160
ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160
7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG",
    }
  ]
}
```

You may notice a few things about this transaction, mostly the things that are missing! Where is Alice's address? Where is Bob's address? Where is the 0.1 input "sent" by Alice? In bitcoin, there are no coins, no senders, no recipients, no balances, no accounts, and no addresses. All those things are constructed at a higher level for the benefit of the user, to make things easier to understand.

You may also notice a lot of strange and indecipherable fields and hexadecimal strings. Don't worry, we will explain each field shown here in detail in this chapter.

Transaction Outputs and Inputs

The fundamental building block of a bitcoin transaction is a *transaction output*. Transaction outputs are indivisible chunks of bitcoin currency, recorded on the blockchain, and recognized as valid by the entire network. Bitcoin full nodes track all available and spendable outputs, known as *unspent transaction outputs*, or *UTXO*. The collection of all UTXO is known as the *UTXO set* and currently numbers in the millions of UTXO. The UTXO set grows as new UTXO is created and shrinks when UTXO is consumed. Every transaction represents a change (state transition) in the UTXO set.

When we say that a user's wallet has "received" bitcoin, what we mean is that the wallet has detected a UTXO that can be spent with one of the keys controlled by that wallet. Thus, a user's bitcoin "balance" is the sum of all UTXO that user's wallet can spend and which may be scattered among hundreds of transactions and hundreds of blocks. The concept of a balance is created by the wallet application. The wallet calculates the user's balance by scanning the blockchain and aggregating the value of any UTXO the wallet can spend with the keys it controls. Most wallets maintain a database or use a database service to store a quick reference set of all the UTXO they can spend with the keys they control.

A transaction output can have an arbitrary (integer) value denominated as a multiple of satoshis. Just like dollars can be divided down to two decimal places as cents, bitcoin can be divided down to eight decimal places as satoshis. Although an output can have any arbitrary value, once created it is indivisible. This is an important character-

istic of outputs that needs to be emphasized: outputs are *discrete* and *indivisible* units of value, denominated in integer satoshis. An unspent output can only be consumed in its entirety by a transaction.

If an UTXO is larger than the desired value of a transaction, it must still be consumed in its entirety and change must be generated in the transaction. In other words, if you have a UTXO worth 20 bitcoin and want to pay only 1 bitcoin, your transaction must consume the entire 20-bitcoin UTXO and produce two outputs: one paying 1 bitcoin to your desired recipient and another paying 19 bitcoin in change back to your wallet. As a result of the indivisible nature of transaction outputs, most bitcoin transactions will have to generate change.

Imagine a shopper buying a $1.50 beverage, reaching into her wallet and trying to find a combination of coins and bank notes to cover the $1.50 cost. The shopper will choose exact change if available e.g. a dollar bill and two quarters (a quarter is $0.25), or a combination of smaller denominations (six quarters), or if necessary, a larger unit such as a $5 note. If she hands too much money, say $5, to the shop owner, she will expect $3.50 change, which she will return to her wallet and have available for future transactions.

Similarly, a bitcoin transaction must be created from a user's UTXO in whatever denominations that user has available. Users cannot cut an UTXO in half any more than they can cut a dollar bill in half and use it as currency. The user's wallet application will typically select from the user's available UTXO to compose an amount greater than or equal to the desired transaction amount.

As with real life, the bitcoin application can use several strategies to satisfy the purchase amount: combining several smaller units, finding exact change, or using a single unit larger than the transaction value and making change. All of this complex assembly of spendable UTXO is done by the user's wallet automatically and is invisible to users. It is only relevant if you are programmatically constructing raw transactions from UTXO.

A transaction consumes previously recorded unspent transaction outputs and creates new transaction outputs that can be consumed by a future transaction. This way, chunks of bitcoin value move forward from owner to owner in a chain of transactions consuming and creating UTXO.

The exception to the output and input chain is a special type of transaction called the *coinbase* transaction, which is the first transaction in each block. This transaction is placed there by the "winning" miner and creates brand-new bitcoin payable to that miner as a reward for mining. This special coinbase transaction does not consume UTXO; instead, it has a special type of input called the "coinbase." This is how bitcoin's money supply is created during the mining process, as we will see in Chapter 10.

What comes first? Inputs or outputs, the chicken or the egg? Strictly speaking, outputs come first because coinbase transactions, which generate new bitcoin, have no inputs and create outputs from nothing.

Transaction Outputs

Every bitcoin transaction creates outputs, which are recorded on the bitcoin ledger. Almost all of these outputs, with one exception (see "Data Recording Output (RETURN)" on page 155) create spendable chunks of bitcoin called UTXO, which are then recognized by the whole network and available for the owner to spend in a future transaction.

UTXO are tracked by every full-node bitcoin client in the UTXO set. New transactions consume (spend) one or more of these outputs from the UTXO set.

Transaction outputs consist of two parts:

- An amount of bitcoin, denominated in *satoshis*, the smallest bitcoin unit
- A cryptographic puzzle that determines the conditions required to spend the output

The cryptographic puzzle is also known as a *locking script*, a *witness script*, or a `scriptPubKey`.

The transaction scripting language, used in the locking script mentioned previously, is discussed in detail in "Transaction Scripts and Script Language" on page 131.

Now, let's look at Alice's transaction (shown previously in "Transactions—Behind the Scenes" on page 118) and see if we can identify the outputs. In the JSON encoding, the outputs are in an array (list) named `vout`:

```
"vout": [
  {
    "value": 0.01500000,
    "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
OP_EQUALVERIFY
    OP_CHECKSIG"
  },
  {
    "value": 0.08450000,
    "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
OP_EQUALVERIFY OP_CHECKSIG",
  }
]
```

As you can see, the transaction contains two outputs. Each output is defined by a value and a cryptographic puzzle. In the encoding shown by Bitcoin Core, the value is

shown in bitcoin, but in the transaction itself it is recorded as an integer denominated in satoshis. The second part of each output is the cryptographic puzzle that sets the conditions for spending. Bitcoin Core shows this as `scriptPubKey` and shows us a human-readable representation of the script.

The topic of locking and unlocking UTXO will be discussed later, in "Script Construction (Lock + Unlock)" on page 132. The scripting language that is used for the script in `scriptPubKey` is discussed in "Transaction Scripts and Script Language" on page 131. But before we delve into those topics, we need to understand the overall structure of transaction inputs and outputs.

Transaction serialization—outputs

When transactions are transmitted over the network or exchanged between applications, they are *serialized*. Serialization is the process of converting the internal representation of a data structure into a format that can be transmitted one byte at a time, also known as a byte stream. Serialization is most commonly used for encoding data structures for transmission over a network or for storage in a file. The serialization format of a transaction output is shown in Table 6-1.

Table 6-1. Transaction output serialization

Size	Field	Description
8 bytes (little-endian)	Amount	Bitcoin value in satoshis (10^{-8} bitcoin)
1–9 bytes (VarInt)	Locking-Script Size	Locking-Script length in bytes, to follow
Variable	Locking-Script	A script defining the conditions needed to spend the output

Most bitcoin libraries and frameworks do not store transactions internally as byte-streams, as that would require complex parsing every time you needed to access a single field. For convenience and readability, bitcoin libraries store transactions internally in data structures (usually object-oriented structures).

The process of converting from the byte-stream representation of a transaction to a library's internal representation data structure is called *deserialization* or *transaction parsing*. The process of converting back to a byte-stream for transmission over the network, for hashing, or for storage on disk is called *serialization*. Most bitcoin libraries have built-in functions for transaction serialization and deserialization.

See if you can manually decode Alice's transaction from the serialized hexadecimal form, finding some of the elements we saw previously. The section containing the two outputs is highlighted in Example 6-1 to help you:

Example 6-1. Alice's transaction, serialized and presented in hexadecimal notation

```
0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73
4d2804fe65fa35779000000008b483045022100884d142d86652a3f47
ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039
ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84
16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1
7b4a10fa336a8d752adfffffffff02 60e31600000000001976a914ab6
8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000
1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac 00000000
```

Here are some hints:

- There are two outputs in the highlighted section, each serialized as shown in Table 6-1.
- The value of 0.015 bitcoin is 1,500,000 satoshis. That's `16 e3 60` in hexadecimal.
- In the serialized transaction, the value `16 e3 60` is encoded in little-endian (least-significant-byte-first) byte order, so it looks like `60 e3 16`.
- The `scriptPubKey` length is 25 bytes, which is `19` in hexadecimal.

Transaction Inputs

Transaction inputs identify (by reference) which UTXO will be consumed and provide proof of ownership through an unlocking script.

To build a transaction, a wallet selects from the UTXO it controls, UTXO with enough value to make the requested payment. Sometimes one UTXO is enough, other times more than one is needed. For each UTXO that will be consumed to make this payment, the wallet creates one input pointing to the UTXO and unlocks it with an unlocking script.

Let's look at the components of an input in greater detail. The first part of an input is a pointer to an UTXO by reference to the transaction hash and sequence number where the UTXO is recorded in the blockchain. The second part is an unlocking script, which the wallet constructs in order to satisfy the spending conditions set in the UTXO. Most often, the unlocking script is a digital signature and public key proving ownership of the bitcoin. However, not all unlocking scripts contain signatures. The third part is a sequence number, which will be discussed later.

Consider our example in "Transactions—Behind the Scenes" on page 118. The transaction inputs are an array (list) called `vin`:

```
"vin": [
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204
b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d1
72787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",
    "sequence": 4294967295
  }
]
```

As you can see, there is only one input in the list (because one UTXO contained sufficient value to make this payment). The input contains four elements:

- A transaction ID, referencing the transaction that contains the UTXO being spent
- An output index (`vout`), identifying which UTXO from that transaction is referenced (first one is zero)
- A `scriptSig`, which satisfies the conditions placed on the UTXO, unlocking it for spending
- A sequence number (to be discussed later)

In Alice's transaction, the input points to the transaction ID:

```
7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18
```

and output index `0` (i.e., the first UTXO created by that transaction). The unlocking script is constructed by Alice's wallet by first retrieving the referenced UTXO, examining its locking script, and then using it to build the necessary unlocking script to satisfy it.

Looking just at the input you may have noticed that we don't know anything about this UTXO, other than a reference to the transaction containing it. We don't know its value (amount in satoshi), and we don't know the locking script that sets the conditions for spending it. To find this information, we must retrieve the referenced UTXO by retrieving the underlying transaction. Notice that because the value of the input is not explicitly stated, we must also use the referenced UTXO in order to calculate the fees that will be paid in this transaction (see "Transaction Fees" on page 126).

It's not just Alice's wallet that needs to retrieve UTXO referenced in the inputs. Once this transaction is broadcast to the network, every validating node will also need to retrieve the UTXO referenced in the transaction inputs in order to validate the transaction.

Transactions on their own seem incomplete because they lack context. They reference UTXO in their inputs but without retrieving that UTXO we cannot know the value of the inputs or their locking conditions. When writing bitcoin software, anytime you decode a transaction with the intent of validating it or counting the fees or checking the unlocking script, your code will first have to retrieve the referenced UTXO from the blockchain in order to build the context implied but not present in the UTXO references of the inputs. For example, to calculate the amount paid in fees, you must know the sum of the values of inputs and outputs. But without retrieving the UTXO referenced in the inputs, you do not know their value. So a seemingly simple operation like counting fees in a single transaction in fact involves multiple steps and data from multiple transactions.

We can use the same sequence of commands with Bitcoin Core as we used when retrieving Alice's transaction (`getrawtransaction` and `decoderawtransaction`). With that we can get the UTXO referenced in the preceding input and take a look:

```
"vout": [
   {
     "value": 0.10000000,
     "scriptPubKey": "OP_DUP OP_HASH160
7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG"
   }
 ]
```

We see that this UTXO has a value of 0.1 BTC and that it has a locking script (`scriptPubKey`) that contains "OP_DUP OP_HASH160…".

To fully understand Alice's transaction we had to retrieve the previous transaction(s) referenced as inputs. A function that retrieves previous transactions and unspent transaction outputs is very common and exists in almost every bitcoin library and API.

Transaction serialization—inputs

When transactions are serialized for transmission on the network, their inputs are encoded into a byte stream as shown in Table 6-2.

Table 6-2. Transaction input serialization

Size	Field	Description
32 bytes	Transaction Hash	Pointer to the transaction containing the UTXO to be spent
4 bytes	Output Index	The index number of the UTXO to be spent; first one is 0
1–9 bytes (VarInt)	Unlocking-Script Size	Unlocking-Script length in bytes, to follow
Variable	Unlocking-Script	A script that fulfills the conditions of the UTXO locking script
4 bytes	Sequence Number	Used for locktime or disabled (0xFFFFFFFF)

As with the outputs, let's see if we can find the inputs from Alice's transaction in the serialized format. First, the inputs decoded:

```
"vin": [
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204
b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d1
72787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",
    "sequence": 4294967295
  }
],
```

Now, let's see if we can identify these fields in the serialized hex encoding in Example 6-2:

Example 6-2. Alice's transaction, serialized and presented in hexadecimal notation

```
0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73
4d2804fe65fa35779000000008b483045022100884d142d86652a3f47
ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039
ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84
16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1
7b4a10fa336a8d752adfffffffff0260e31600000000001976a914ab6
8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000
1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000 000
```

Hints:

- The transaction ID is serialized in reversed byte order, so it starts with (hex) `18` and ends with `79`
- The output index is a 4-byte group of zeros, easy to identify
- The length of the `scriptSig` is 139 bytes, or `8b` in hex
- The sequence number is set to `FFFFFFFF`, again easy to identify

Transaction Fees

Most transactions include transaction fees, which compensate the bitcoin miners for securing the network. Fees also serve as a security mechanism themselves, by making it economically infeasible for attackers to flood the network with transactions. Min-

ing and the fees and rewards collected by miners are discussed in more detail in Chapter 10.

This section examines how transaction fees are included in a typical transaction. Most wallets calculate and include transaction fees automatically. However, if you are constructing transactions programmatically, or using a command-line interface, you must manually account for and include these fees.

Transaction fees serve as an incentive to include (mine) a transaction into the next block and also as a disincentive against abuse of the system by imposing a small cost on every transaction. Transaction fees are collected by the miner who mines the block that records the transaction on the blockchain.

Transaction fees are calculated based on the size of the transaction in kilobytes, not the value of the transaction in bitcoin. Overall, transaction fees are set based on market forces within the bitcoin network. Miners prioritize transactions based on many different criteria, including fees, and might even process transactions for free under certain circumstances. Transaction fees affect the processing priority, meaning that a transaction with sufficient fees is likely to be included in the next block mined, whereas a transaction with insufficient or no fees might be delayed, processed on a best-effort basis after a few blocks, or not processed at all. Transaction fees are not mandatory, and transactions without fees might be processed eventually; however, including transaction fees encourages priority processing.

Over time, the way transaction fees are calculated and the effect they have on transaction prioritization has evolved. At first, transaction fees were fixed and constant across the network. Gradually, the fee structure relaxed and may be influenced by market forces, based on network capacity and transaction volume. Since at least the beginning of 2016, capacity limits in bitcoin have created competition between transactions, resulting in higher fees and effectively making free transactions a thing of the past. Zero fee or very low fee transactions rarely get mined and sometimes will not even be propagated across the network.

In Bitcoin Core, fee relay policies are set by the `minrelaytxfee` option. The current default `minrelaytxfee` is 0.00001 bitcoin or a hundredth of a millibitcoin per kilobyte. Therefore, by default, transactions with a fee less than 0.0001 bitcoin are treated as free and are only relayed if there is space in the mempool; otherwise, they are dropped. Bitcoin nodes can override the default fee relay policy by adjusting the value of `minrelaytxfee`.

Any bitcoin service that creates transactions, including wallets, exchanges, retail applications, etc., *must* implement dynamic fees. Dynamic fees can be implemented through a third-party fee estimation service or with a built-in fee estimation algorithm. If you're unsure, begin with a third-party service and as you gain experience

design and implement your own algorithm if you wish to remove the third-party dependency.

Fee estimation algorithms calculate the appropriate fee, based on capacity and the fees offered by "competing" transactions. These algorithms range from simplistic (average or median fee in the last block) to sophisticated (statistical analysis). They estimate the necessary fee (in satoshis per byte) that will give a transaction a high probability of being selected and included within a certain number of blocks. Most services offer users the option of choosing high, medium, or low priority fees. High priority means users pay higher fees but the transaction is likely to be included in the next block. Medium and low priority means users pay lower transaction fees but the transactions may take much longer to confirm.

Many wallet applications use third-party services for fee calculations. One popular service is *http://bitcoinfees.21.co*, which provides an API and a visual chart showing the fee in satoshi/byte for different priorities.

Static fees are no longer viable on the bitcoin network. Wallets that set static fees will produce a poor user experience as transactions will often get "stuck" and remain unconfirmed. Users who don't understand bitcoin transactions and fees are dismayed by "stuck" transactions because they think they've lost their money.

The chart in Figure 6-2 shows the real-time estimate of fees in 10 satoshi/byte increments and the expected confirmation time (in minutes and number of blocks) for transactions with fees in each range. For each fee range (e.g., 61–70 satoshi/byte), two horizontal bars show the number of unconfirmed transactions (1405) and total number of transactions in the past 24 hours (102,975), with fees in that range. Based on the graph, the recommended high-priority fee at this time was 80 satoshi/byte, a fee likely to result in the transaction being mined in the very next block (zero block delay). For perspective, the median transaction size is 226 bytes, so the recommended fee for a transaction size would be 18,080 satoshis (0.00018080 BTC).

The fee estimation data can be retrieved via a simple HTTP REST API, at *https://bitcoinfees.21.co/api/v1/fees/recommended*. For example, on the command line using the `curl` command:

```
$ curl https://bitcoinfees.21.co/api/v1/fees/recommended

{"fastestFee":80,"halfHourFee":80,"hourFee":60}
```

The API returns a JSON object with the current fee estimate for fastest confirmation (`fastestFee`), confirmation within three blocks (`halfHourFee`) and six blocks (`hourFee`), in satoshi per byte.

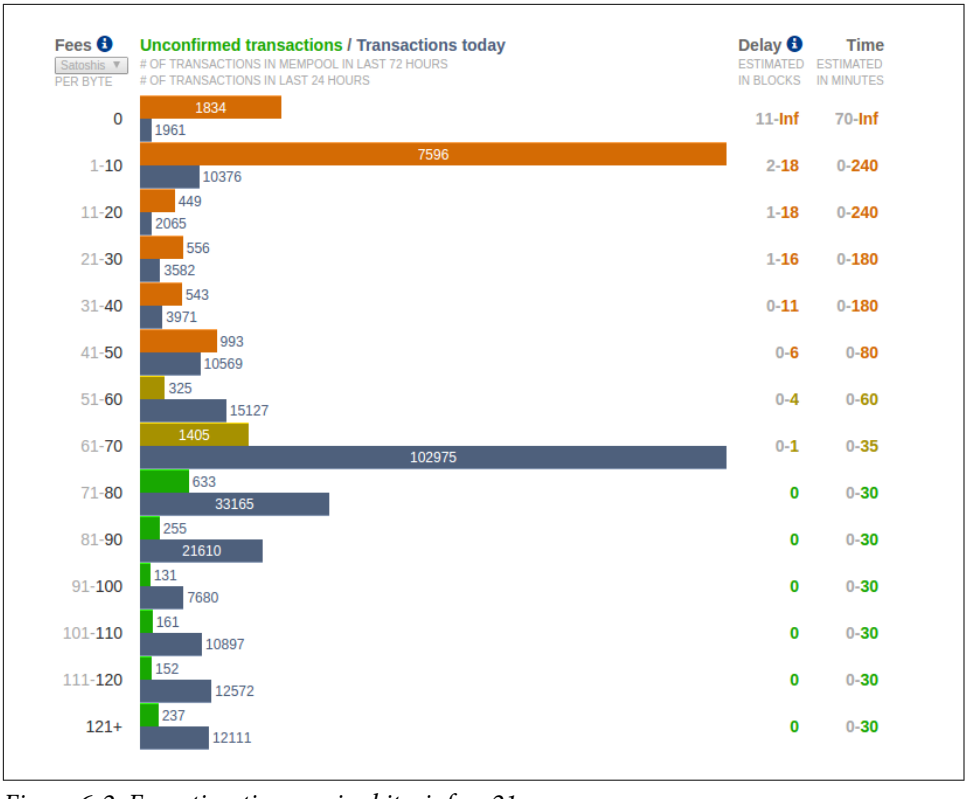

Figure 6-2. Fee estimation service bitcoinfees.21.co

Adding Fees to Transactions

The data structure of transactions does not have a field for fees. Instead, fees are implied as the difference between the sum of inputs and the sum of outputs. Any excess amount that remains after all outputs have been deducted from all inputs is the fee that is collected by the miners:

```
Fees = Sum(Inputs) - Sum(Outputs)
```

This is a somewhat confusing element of transactions and an important point to understand, because if you are constructing your own transactions you must ensure you do not inadvertently include a very large fee by underspending the inputs. That means that you must account for all inputs, if necessary by creating change, or you will end up giving the miners a very big tip!

For example, if you consume a 20-bitcoin UTXO to make a 1-bitcoin payment, you must include a 19-bitcoin change output back to your wallet. Otherwise, the 19-bitcoin "leftover" will be counted as a transaction fee and will be collected by the

miner who mines your transaction in a block. Although you will receive priority processing and make a miner very happy, this is probably not what you intended.

If you forget to add a change output in a manually constructed transaction, you will be paying the change as a transaction fee. "Keep the change!" might not be what you intended.

Let's see how this works in practice, by looking at Alice's coffee purchase again. Alice wants to spend 0.015 bitcoin to pay for coffee. To ensure this transaction is processed promptly, she will want to include a transaction fee, say 0.001. That will mean that the total cost of the transaction will be 0.016. Her wallet must therefore source a set of UTXO that adds up to 0.016 bitcoin or more and, if necessary, create change. Let's say her wallet has a 0.2-bitcoin UTXO available. It will therefore need to consume this UTXO, create one output to Bob's Cafe for 0.015, and a second output with 0.184 bitcoin in change back to her own wallet, leaving 0.001 bitcoin unallocated, as an implicit fee for the transaction.

Now let's look at a different scenario. Eugenia, our children's charity director in the Philippines, has completed a fundraiser to purchase schoolbooks for the children. She received several thousand small donations from people all around the world, totaling 50 bitcoin, so her wallet is full of very small payments (UTXO). Now she wants to purchase hundreds of schoolbooks from a local publisher, paying in bitcoin.

As Eugenia's wallet application tries to construct a single larger payment transaction, it must source from the available UTXO set, which is composed of many smaller amounts. That means that the resulting transaction will source from more than a hundred small-value UTXO as inputs and only one output, paying the book publisher. A transaction with that many inputs will be larger than one kilobyte, perhaps a kilobyte or several kilobytes in size. As a result, it will require a much higher fee than the median-sized transaction.

Eugenia's wallet application will calculate the appropriate fee by measuring the size of the transaction and multiplying that by the per-kilobyte fee. Many wallets will overpay fees for larger transactions to ensure the transaction is processed promptly. The higher fee is not because Eugenia is spending more money, but because her transaction is more complex and larger in size—the fee is independent of the transaction's bitcoin value.

Transaction Scripts and Script Language

The bitcoin transaction script language, called *Script*, is a Forth-like reverse-polish notation stack-based execution language. If that sounds like gibberish, you probably haven't studied 1960s programming languages, but that's ok—we will explain it all in this chapter. Both the locking script placed on a UTXO and the unlocking script are written in this scripting language. When a transaction is validated, the unlocking script in each input is executed alongside the corresponding locking script to see if it satisfies the spending condition.

Script is a very simple language that was designed to be limited in scope and executable on a range of hardware, perhaps as simple as an embedded device. It requires minimal processing and cannot do many of the fancy things modern programming languages can do. For its use in validating programmable money, this is a deliberate security feature.

Today, most transactions processed through the bitcoin network have the form "Payment to Bob's bitcoin address" and are based on a script called a Pay-to-Public-Key-Hash script. However, bitcoin transactions are not limited to the "Payment to Bob's bitcoin address" script. In fact, locking scripts can be written to express a vast variety of complex conditions. In order to understand these more complex scripts, we must first understand the basics of transaction scripts and script language.

In this section, we will demonstrate the basic components of the bitcoin transaction scripting language and show how it can be used to express simple conditions for spending and how those conditions can be satisfied by unlocking scripts.

Bitcoin transaction validation is not based on a static pattern, but instead is achieved through the execution of a scripting language. This language allows for a nearly infinite variety of conditions to be expressed. This is how bitcoin gets the power of "programmable money."

Turing Incompleteness

The bitcoin transaction script language contains many operators, but is deliberately limited in one important way—there are no loops or complex flow control capabilities other than conditional flow control. This ensures that the language is not *Turing Complete*, meaning that scripts have limited complexity and predictable execution times. Script is not a general-purpose language. These limitations ensure that the language cannot be used to create an infinite loop or other form of "logic bomb" that could be embedded in a transaction in a way that causes a denial-of-service attack against the bitcoin network. Remember, every transaction is validated by every full

node on the bitcoin network. A limited language prevents the transaction validation mechanism from being used as a vulnerability.

Stateless Verification

The bitcoin transaction script language is stateless, in that there is no state prior to execution of the script, or state saved after execution of the script. Therefore, all the information needed to execute a script is contained within the script. A script will predictably execute the same way on any system. If your system verifies a script, you can be sure that every other system in the bitcoin network will also verify the script, meaning that a valid transaction is valid for everyone and everyone knows this. This predictability of outcomes is an essential benefit of the bitcoin system.

Script Construction (Lock + Unlock)

Bitcoin's transaction validation engine relies on two types of scripts to validate transactions: a locking script and an unlocking script.

A locking script is a spending condition placed on an output: it specifies the conditions that must be met to spend the output in the future. Historically, the locking script was called a *scriptPubKey*, because it usually contained a public key or bitcoin address (public key hash). In this book we refer to it as a "locking script" to acknowledge the much broader range of possibilities of this scripting technology. In most bitcoin applications, what we refer to as a locking script will appear in the source code as `scriptPubKey`. You will also see the locking script referred to as a *witness script* (see Appendix D) or more generally as a *cryptographic puzzle*. These terms all mean the same thing, at different levels of abstraction.

An unlocking script is a script that "solves," or satisfies, the conditions placed on an output by a locking script and allows the output to be spent. Unlocking scripts are part of every transaction input. Most of the time they contain a digital signature produced by the user's wallet from his or her private key. Historically, the unlocking script was called *scriptSig*, because it usually contained a digital signature. In most bitcoin applications, the source code refers to the unlocking script as `scriptSig`. You will also see the unlocking script referred to as a *witness* (see Appendix D). In this book, we refer to it as an "unlocking script" to acknowledge the much broader range of locking script requirements, because not all unlocking scripts must contain signatures.

Every bitcoin validating node will validate transactions by executing the locking and unlocking scripts together. Each input contains an unlocking script and refers to a previously existing UTXO. The validation software will copy the unlocking script, retrieve the UTXO referenced by the input, and copy the locking script from that UTXO. The unlocking and locking script are then executed in sequence. The input is valid if the unlocking script satisfies the locking script conditions (see "Separate exe-

cution of unlocking and locking scripts" on page 136). All the inputs are validated independently, as part of the overall validation of the transaction.

Note that the UTXO is permanently recorded in the blockchain, and therefore is invariable and is unaffected by failed attempts to spend it by reference in a new transaction. Only a valid transaction that correctly satisfies the conditions of the output results in the output being considered as "spent" and removed from the set of unspent transaction outputs (UTXO set).

Figure 6-3 is an example of the unlocking and locking scripts for the most common type of bitcoin transaction (a payment to a public key hash), showing the combined script resulting from the concatenation of the unlocking and locking scripts prior to script validation.

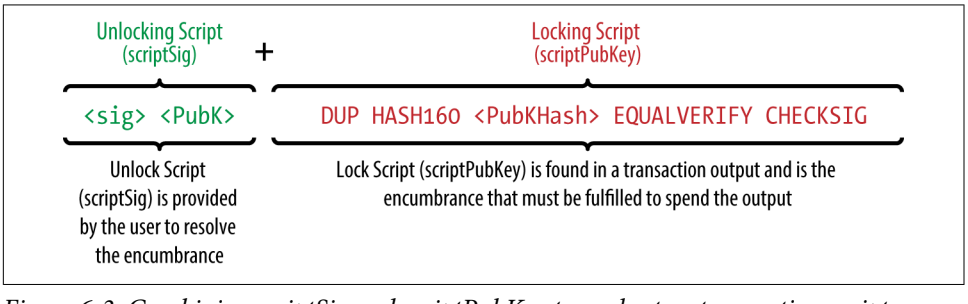

Figure 6-3. Combining scriptSig and scriptPubKey to evaluate a transaction script

The script execution stack

Bitcoin's scripting language is called a stack-based language because it uses a data structure called a *stack*. A stack is a very simple data structure that can be visualized as a stack of cards. A stack allows two operations: push and pop. Push adds an item on top of the stack. Pop removes the top item from the stack. Operations on a stack can only act on the topmost item on the stack. A stack data structure is also called a Last-In-First-Out, or "LIFO" queue.

The scripting language executes the script by processing each item from left to right. Numbers (data constants) are pushed onto the stack. Operators push or pop one or more parameters from the stack, act on them, and might push a result onto the stack. For example, `OP_ADD` will pop two items from the stack, add them, and push the resulting sum onto the stack.

Conditional operators evaluate a condition, producing a boolean result of TRUE or FALSE. For example, `OP_EQUAL` pops two items from the stack and pushes TRUE (TRUE is represented by the number 1) if they are equal or FALSE (represented by zero) if they are not equal. Bitcoin transaction scripts usually contain a conditional operator, so that they can produce the TRUE result that signifies a valid transaction.

A simple script

Now let's apply what we've learned about scripts and stacks to some simple examples.

In Figure 6-4, the script `2 3 OP_ADD 5 OP_EQUAL` demonstrates the arithmetic addition operator `OP_ADD`, adding two numbers and putting the result on the stack, followed by the conditional operator `OP_EQUAL`, which checks that the resulting sum is equal to `5`. For brevity, the `OP_` prefix is omitted in the step-by-step example. For more details on the available script operators and functions, see Appendix B.

Although most locking scripts refer to a public key hash (essentially, a bitcoin address), thereby requiring proof of ownership to spend the funds, the script does not have to be that complex. Any combination of locking and unlocking scripts that results in a TRUE value is valid. The simple arithmetic we used as an example of the scripting language is also a valid locking script that can be used to lock a transaction output.

Use part of the arithmetic example script as the locking script:

```
3 OP_ADD 5 OP_EQUAL
```

which can be satisfied by a transaction containing an input with the unlocking script:

```
2
```

The validation software combines the locking and unlocking scripts and the resulting script is:

```
2 3 OP_ADD 5 OP_EQUAL
```

As we saw in the step-by-step example in Figure 6-4, when this script is executed, the result is `OP_TRUE`, making the transaction valid. Not only is this a valid transaction output locking script, but the resulting UTXO could be spent by anyone with the arithmetic skills to know that the number 2 satisfies the script.

Transactions are valid if the top result on the stack is `TRUE` (noted as `{0x01}`), any other nonzero value, or if the stack is empty after script execution. Transactions are invalid if the top value on the stack is `FALSE` (a zero-length empty value, noted as `{}`) or if script execution is halted explicitly by an operator, such as `OP_VERIFY`, `OP_RETURN`, or a conditional terminator such as `OP_ENDIF`. See Appendix B for details.

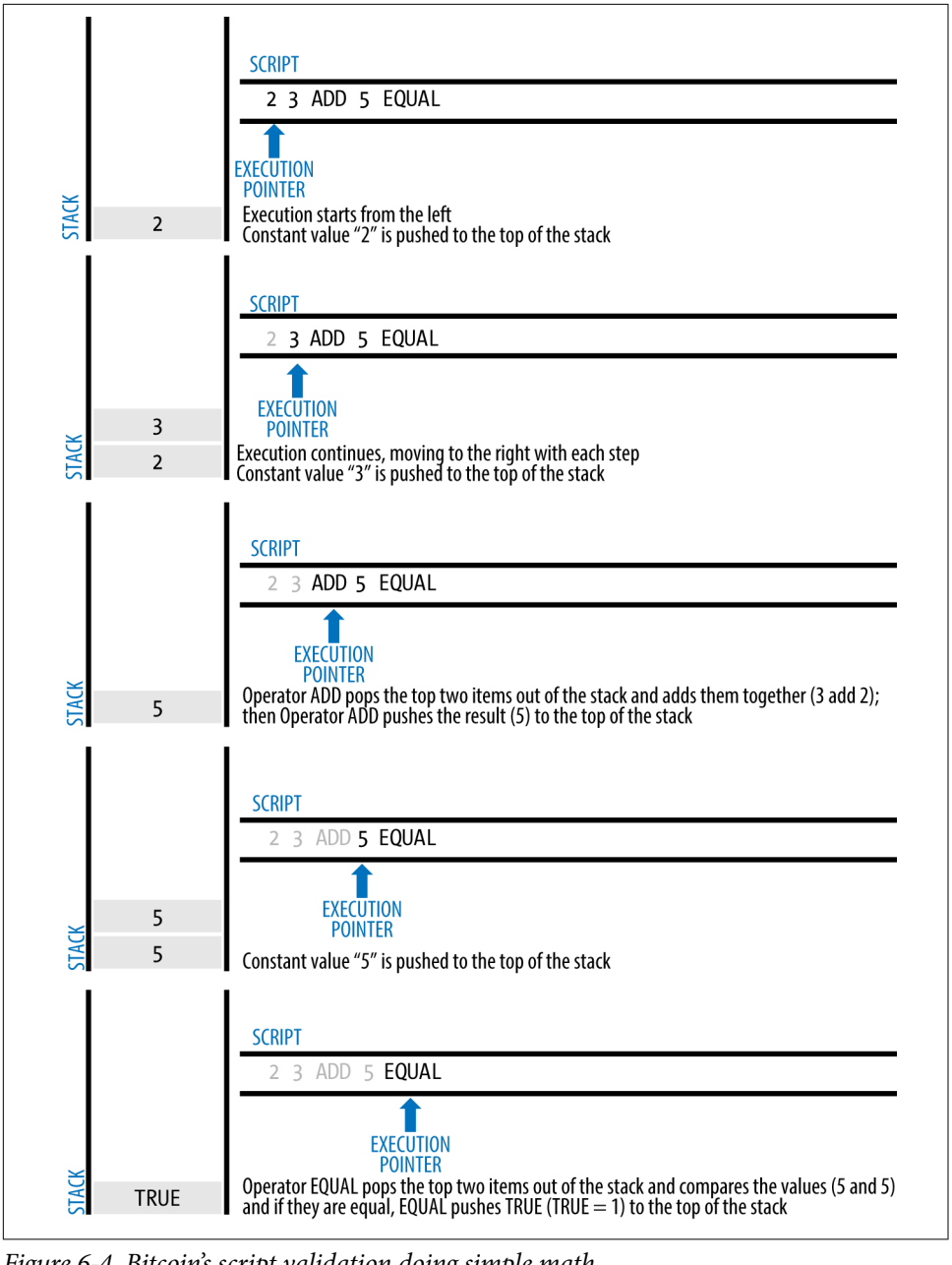

Figure 6-4. Bitcoin's script validation doing simple math

The following is a slightly more complex script, which calculates `2 + 7 − 3 + 1`. Notice that when the script contains several operators in a row, the stack allows the results of one operator to be acted upon by the next operator:

```
2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL
```

Try validating the preceding script yourself using pencil and paper. When the script execution ends, you should be left with the value `TRUE` on the stack.

Separate execution of unlocking and locking scripts

In the original bitcoin client, the unlocking and locking scripts were concatenated and executed in sequence. For security reasons, this was changed in 2010, because of a vulnerability that allowed a malformed unlocking script to push data onto the stack and corrupt the locking script. In the current implementation, the scripts are executed separately with the stack transferred between the two executions, as described next.

First, the unlocking script is executed, using the stack execution engine. If the unlocking script is executed without errors (e.g., it has no "dangling" operators left over), the main stack (not the alternate stack) is copied and the locking script is executed. If the result of executing the locking script with the stack data copied from the unlocking script is "TRUE," the unlocking script has succeeded in resolving the conditions imposed by the locking script and, therefore, the input is a valid authorization to spend the UTXO. If any result other than "TRUE" remains after execution of the combined script, the input is invalid because it has failed to satisfy the spending conditions placed on the UTXO.

Pay-to-Public-Key-Hash (P2PKH)

The vast majority of transactions processed on the bitcoin network spend outputs locked with a Pay-to-Public-Key-Hash or "P2PKH" script. These outputs contain a locking script that locks the output to a public key hash, more commonly known as a bitcoin address. An output locked by a P2PKH script can be unlocked (spent) by presenting a public key and a digital signature created by the corresponding private key (see "Digital Signatures (ECDSA)" on page 138).

For example, let's look at Alice's payment to Bob's Cafe again. Alice made a payment of 0.015 bitcoin to the cafe's bitcoin address. That transaction output would have a locking script of the form:

```
OP_DUP OP_HASH160 <Cafe Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

The `Cafe Public Key Hash` is equivalent to the bitcoin address of the cafe, without the Base58Check encoding. Most applications would show the *public key hash* in hexadecimal encoding and not the familiar bitcoin address Base58Check format that begins with a "1."

The preceding locking script can be satisfied with an unlocking script of the form:

```
<Cafe Signature> <Cafe Public Key>
```

The two scripts together would form the following combined validation script:

```
<Cafe Signature> <Cafe Public Key> OP_DUP OP_HASH160
<Cafe Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

When executed, this combined script will evaluate to TRUE if, and only if, the unlocking script matches the conditions set by the locking script. In other words, the result will be TRUE if the unlocking script has a valid signature from the cafe's private key that corresponds to the public key hash set as an encumbrance.

Figures 6-5 and 6-6 show (in two parts) a step-by-step execution of the combined script, which will prove this is a valid transaction.

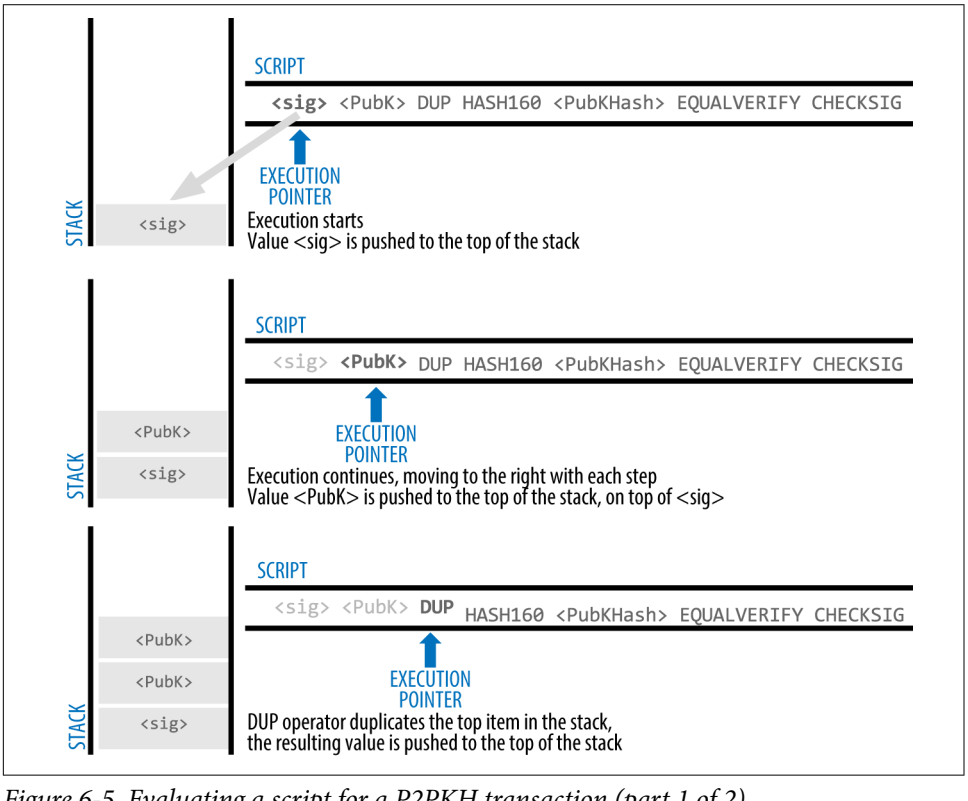

Figure 6-5. Evaluating a script for a P2PKH transaction (part 1 of 2)

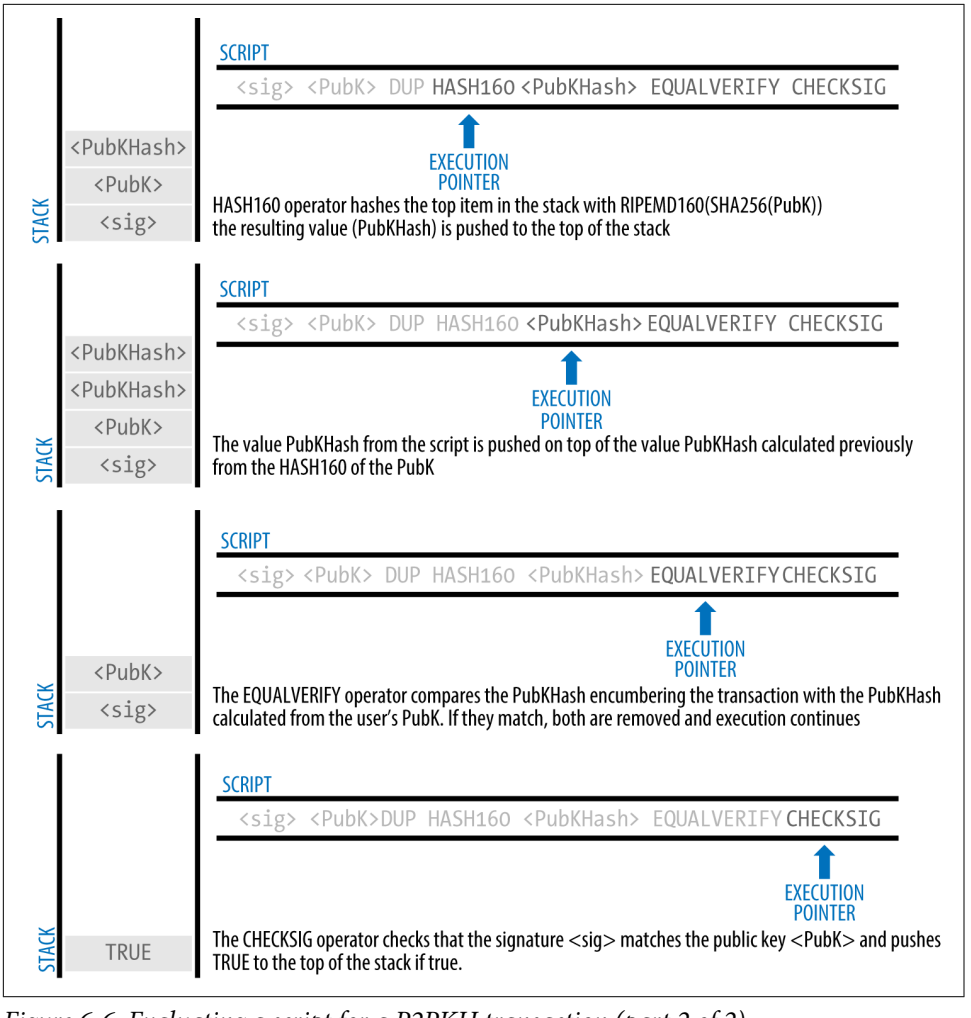

Figure 6-6. Evaluating a script for a P2PKH transaction (part 2 of 2)

Digital Signatures (ECDSA)

So far, we have not delved into any detail about "digital signatures." In this section we look at how digital signatures work and how they can present proof of ownership of a private key without revealing that private key.

The digital signature algorithm used in bitcoin is the *Elliptic Curve Digital Signature Algorithm*, or *ECDSA*. ECDSA is the algorithm used for digital signatures based on elliptic curve private/public key pairs, as described in "Elliptic Curve Cryptography Explained" on page 60. ECDSA is used by the script functions `OP_CHECKSIG`, `OP_CHECKSIGVERIFY`, `OP_CHECKMULTISIG`, and `OP_CHECKMULTISIGVERIFY`. Any time

you see those in a locking script, the unlocking script must contain an ECDSA signature.

A digital signature serves three purposes in bitcoin (see the following sidebar). First, the signature proves that the owner of the private key, who is by implication the owner of the funds, has *authorized* the spending of those funds. Secondly, the proof of authorization is *undeniable* (nonrepudiation). Thirdly, the signature proves that the transaction (or specific parts of the transaction) have not and *cannot be modified* by anyone after it has been signed.

Note that each transaction input is signed independently. This is critical, as neither the signatures nor the inputs have to belong to or be applied by the same "owners." In fact, a specific transaction scheme called "CoinJoin" uses this fact to create multi-party transactions for privacy.

Each transaction input and any signature it may contain is *completely* independent of any other input or signature. Multiple parties can collaborate to construct transactions and sign only one input each.

Wikipedia's Definition of a "Digital Signature"

A digital signature is a mathematical scheme for demonstrating the authenticity of a digital message or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication), that the sender cannot deny having sent the message (nonrepudiation), and that the message was not altered in transit (integrity).

Source: https://en.wikipedia.org/wiki/Digital_signature

How Digital Signatures Work

A digital signature is a *mathematical scheme* that consists of two parts. The first part is an algorithm for creating a signature, using a private key (the signing key), from a message (the transaction). The second part is an algorithm that allows anyone to verify the signature, given also the message and a public key.

Creating a digital signature

In bitcoin's implementation of the ECDSA algorithm, the "message" being signed is the transaction, or more accurately a hash of a specific subset of the data in the transaction (see "Signature Hash Types (SIGHASH)" on page 141). The signing key is the user's private key. The result is the signature:

$$Sig = F_{sig}(F_{hash}(m), dA)$$

where:

- dA is the signing private key
- m is the transaction (or parts of it)
- F_{hash} is the hashing function
- F_{sig} is the signing algorithm
- Sig is the resulting signature

More details on the mathematics of ECDSA can be found in "ECDSA Math" on page 143.

The function F_{sig} produces a signature `Sig` that is composed of two values, commonly referred to as `R` and `S`:

```
Sig = (R, S)
```

Now that the two values `R` and `S` have been calculated, they are serialized into a byte-stream using an international standard encoding scheme called the *Distinguished Encoding Rules*, or *DER*.

Serialization of signatures (DER)

Let's look at the transaction Alice created again. In the transaction input there is an unlocking script that contains the following DER-encoded signature from Alice's wallet:

```
3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b
9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301
```

That signature is a serialized byte-stream of the `R` and `S` values produced by Alice's wallet to prove she owns the private key authorized to spend that output. The serialization format consists of nine elements as follows:

- `0x30`—indicating the start of a DER sequence
- `0x45`—the length of the sequence (69 bytes)
- `0x02`—an integer value follows
- `0x21`—the length of the integer (33 bytes)
- `R`—`00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb`
- `0x02`—another integer follows
- `0x20`—the length of the integer (32 bytes)

- `S`—`4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813`
- A suffix (`0x01`) indicating the type of hash used (`SIGHASH_ALL`)

See if you can decode Alice's serialized (DER-encoded) signature using this list. The important numbers are `R` and `S`; the rest of the data is part of the DER encoding scheme.

Verifying the Signature

To verify the signature, one must have the signature (`R` and `S`), the serialized transaction, and the public key (that corresponds to the private key used to create the signature). Essentially, verification of a signature means "Only the owner of the private key that generated this public key could have produced this signature on this transaction."

The signature verification algorithm takes the message (a hash of the transaction or parts of it), the signer's public key and the signature (`R` and `S` values), and returns TRUE if the signature is valid for this message and public key.

Signature Hash Types (SIGHASH)

Digital signatures are applied to messages, which in the case of bitcoin, are the transactions themselves. The signature implies a *commitment* by the signer to specific transaction data. In the simplest form, the signature applies to the entire transaction, thereby committing all the inputs, outputs, and other transaction fields. However, a signature can commit to only a subset of the data in a transaction, which is useful for a number of scenarios as we will see in this section.

Bitcoin signatures have a way of indicating which part of a transaction's data is included in the hash signed by the private key using a `SIGHASH` flag. The `SIGHASH` flag is a single byte that is appended to the signature. Every signature has a `SIGHASH` flag and the flag can be different from to input to input. A transaction with three signed inputs may have three signatures with different `SIGHASH` flags, each signature signing (committing) different parts of the transaction.

Remember, each input may contain a signature in its unlocking script. As a result, a transaction that contains several inputs may have signatures with different `SIGHASH` flags that commit different parts of the transaction in each of the inputs. Note also that bitcoin transactions may contain inputs from different "owners," who may sign only one input in a partially constructed (and invalid) transaction, collaborating with others to gather all the necessary signatures to make a valid transaction. Many of the `SIGHASH` flag types only make sense if you think of multiple participants collaborating outside the bitcoin network and updating a partially signed transaction.

There are three SIGHASH flags: ALL, NONE, and SINGLE, as shown in Table 6-3.

Table 6-3. SIGHASH types and their meanings

SIGHASH flag	Value	Description
ALL	0x01	Signature applies to all inputs and outputs
NONE	0x02	Signature applies to all inputs, none of the outputs
SINGLE	0x03	Signature applies to all inputs but only the one output with the same index number as the signed input

In addition, there is a modifier flag SIGHASH_ANYONECANPAY, which can be combined with each of the preceding flags. When ANYONECANPAY is set, only one input is signed, leaving the rest (and their sequence numbers) open for modification. The ANYONECANPAY has the value 0x80 and is applied by bitwise OR, resulting in the combined flags as shown in Table 6-4.

Table 6-4. SIGHASH types with modifiers and their meanings

SIGHASH flag	Value	Description
ALL|ANYONECANPAY	0x81	Signature applies to one inputs and all outputs
NONE|ANYONECANPAY	0x82	Signature applies to one inputs, none of the outputs
SINGLE|ANYONECANPAY	0x83	Signature applies to one input and the output with the same index number

The way SIGHASH flags are applied during signing and verification is that a copy of the transaction is made and certain fields within are truncated (set to zero length and emptied). The resulting transaction is serialized. The SIGHASH flag is added to the end of the serialized transaction and the result is hashed. The hash itself is the "message" that is signed. Depending on which SIGHASH flag is used, different parts of the transaction are truncated. The resulting hash depends on different subsets of the data in the transaction. By including the SIGHASH as the last step before hashing, the signature commits the SIGHASH type as well, so it can't be changed (e.g., by a miner).

All SIGHASH types sign the transaction nLocktime field (see "Transaction Locktime (nLocktime)" on page 157). In addition, the SIGHASH type itself is appended to the transaction before it is signed, so that it can't be modified once signed.

In the example of Alice's transaction (see the list in "Serialization of signatures (DER)" on page 140), we saw that the last part of the DER-encoded signature was 01, which is the SIGHASH_ALL flag. This locks the transaction data, so Alice's signature is committing the state of all inputs and outputs. This is the most common signature form.

Let's look at some of the other SIGHASH types and how they can be used in practice:

ALL|ANYONECANPAY
: This construction can be used to make a "crowdfunding"-style transaction. Someone attempting to raise funds can construct a transaction with a single output. The single output pays the "goal" amount to the fundraiser. Such a transaction is obviously not valid, as it has no inputs. However, others can now amend it by adding an input of their own, as a donation. They sign their own input with ALL|ANYONECANPAY. Unless enough inputs are gathered to reach the value of the output, the transaction is invalid. Each donation is a "pledge," which cannot be collected by the fundraiser until the entire goal amount is raised.

NONE
: This construction can be used to create a "bearer check" or "blank check" of a specific amount. It commits to the input, but allows the output locking script to be changed. Anyone can write their own bitcoin address into the output locking script and redeem the transaction. However, the output value itself is locked by the signature.

NONE|ANYONECANPAY
: This construction can be used to build a "dust collector." Users who have tiny UTXO in their wallets can't spend these without the cost in fees exceeding the value of the dust. With this type of signature, the dust UTXO can be donated for anyone to aggregate and spend whenever they want.

There are some proposals to modify or expand the SIGHASH system. One such proposal is *Bitmask Sighash Modes* by Blockstream's Glenn Willen, as part of the Elements project. This aims to create a flexible replacement for SIGHASH types that allows "arbitrary, miner-rewritable bitmasks of inputs and outputs" that can express "more complex contractual precommitment schemes, such as signed offers with change in a distributed asset exchange."

You will not see SIGHASH flags presented as an option in a user's wallet application. With few exceptions, wallets construct P2PKH scripts and sign with SIGHASH_ALL flags. To use a different SIGHASH flag, you would have to write software to construct and sign transactions. More importantly, SIGHASH flags can be used by special-purpose bitcoin applications that enable novel uses.

ECDSA Math

As mentioned previously, signatures are created by a mathematical function F_{sig} that produces a signature composed of two values R and S. In this section we look at the function F_{sig} in more detail.

The signature algorithm first generates an *ephemeral* (temporary) private public key pair. This temporary key pair is used in the calculation of the *R* and *S* values, after a transformation involving the signing private key and the transaction hash.

The temporary key pair is based on a random number *k*, which is used as the temporary private key. From *k*, we generate the corresponding temporary public key *P* (calculated as $P = k*G$, in the same way bitcoin public keys are derived; see "Public Keys" on page 60). The *R* value of the digital signature is then the x coordinate of the ephemeral public key *P*.

From there, the algorithm calculates the *S* value of the signature, such that:

$S = k^{-1} (Hash(m) + dA * R) \ mod \ p$

where:

- *k* is the ephemeral private key
- *R* is the x coordinate of the ephemeral public key
- *dA* is the signing private key
- *m* is the transaction data
- *p* is the prime order of the elliptic curve

Verification is the inverse of the signature generation function, using the *R*, *S* values and the public key to calculate a value *P*, which is a point on the elliptic curve (the ephemeral public key used in signature creation):

$P = S^{-1} * Hash(m) * G + S^{-1} * R * Qa$

where:

- *R* and *S* are the signature values
- *Qa* is Alice's public key
- *m* is the transaction data that was signed
- *G* is the elliptic curve generator point

If the x coordinate of the calculated point *P* is equal to *R*, then the verifier can conclude that the signature is valid.

Note that in verifying the signature, the private key is neither known nor revealed.

The math of ECDSA is complex and difficult to understand. There are a number of great guides online that might help. Search for "ECDSA explained" or try this one: *http://bit.ly/2r0HhGB*.

The Importance of Randomness in Signatures

As we saw in "ECDSA Math" on page 143, the signature generation algorithm uses a random key *k*, as the basis for an ephemeral private/public key pair. The value of *k* is not important, *as long as it is random*. If the same value *k* is used to produce two signatures on different messages (transactions), then the signing *private key* can be calculated by anyone. Reuse of the same value for *k* in a signature algorithm leads to exposure of the private key!

If the same value *k* is used in the signing algorithm on two different transactions, the private key can be calculated and exposed to the world!

This is not just a theoretical possibility. We have seen this issue lead to exposure of private keys in a few different implementations of transaction-signing algorithms in bitcoin. People have had funds stolen because of inadvertent reuse of a *k* value. The most common reason for reuse of a *k* value is an improperly initialized random-number generator.

To avoid this vulnerability, the industry best practice is to not generate *k* with a random-number generator seeded with entropy, but instead to use a deterministic-random process seeded with the transaction data itself. This ensures that each transaction produces a different *k*. The industry-standard algorithm for deterministic initialization of *k* is defined in RFC 6979, published by the Internet Engineering Task Force.

If you are implementing an algorithm to sign transactions in bitcoin, you *must* use RFC 6979 or a similarly deterministic-random algorithm to ensure you generate a different *k* for each transaction.

Bitcoin Addresses, Balances, and Other Abstractions

We began this chapter with the discovery that transactions look very different "behind the scenes" than how they are presented in wallets, blockchain explorers, and other user-facing applications. Many of the simplistic and familiar concepts from the earlier chapters, such as bitcoin addresses and balances, seem to be absent from the transaction structure. We saw that transactions don't contain bitcoin addresses, per se, but instead operate through scripts that lock and unlock discrete values of bitcoin. Balances are not present anywhere in this system and yet every wallet application prominently displays the balance of the user's wallet.

Now that we have explored what is actually included in a bitcoin transaction, we can examine how the higher-level abstractions are derived from the seemingly primitive components of the transaction.

Let's look again at how Alice's transaction was presented on a popular block explorer (Figure 6-7).

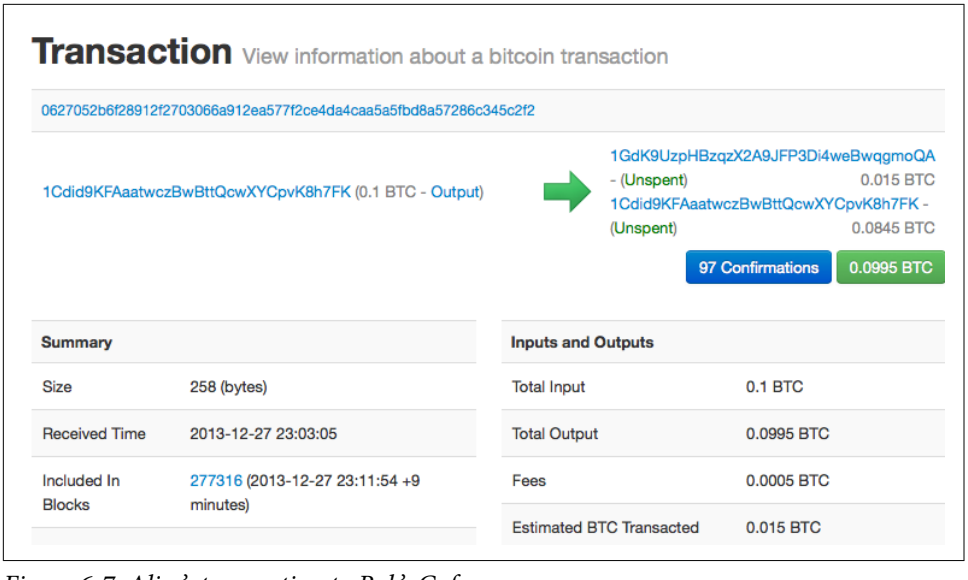

Figure 6-7. Alice's transaction to Bob's Cafe

On the left side of the transaction, the blockchain explorer shows Alice's bitcoin address as the "sender." In fact, this information is not in the transaction itself. When the blockchain explorer retrieved the transaction it also retrieved the previous transaction referenced in the input and extracted the first output from that older transaction. Within that output is a locking script that locks the UTXO to Alice's public key hash (a P2PKH script). The blockchain explorer extracted the public key hash and encoded it using Base58Check encoding to produce and display the bitcoin address that represents that public key.

Similarly, on the right side, the blockchain explorer shows the two outputs; the first to Bob's bitcoin address and the second to Alice's bitcoin address (as change). Once again, to create these bitcoin addresses, the blockchain explorer extracted the locking script from each output, recognized it as a P2PKH script, and extracted the public-key-hash from within. Finally, the blockchain explorer reencoded that public key hash with Base58Check to produce and display the bitcoin addresses.

If you were to click on Bob's bitcoin address, the blockchain explorer would show you the view in Figure 6-8.

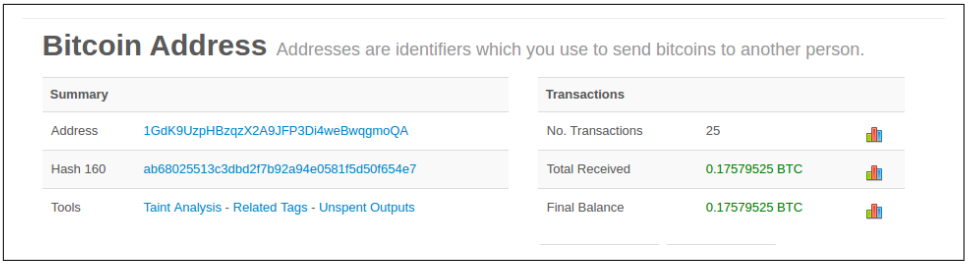

Figure 6-8. The balance of Bob's bitcoin address

The blockchain explorer displays the balance of Bob's bitcoin address. But nowhere in the bitcoin system is there a concept of a "balance." Rather, the values displayed here are constructed by the blockchain explorer as follows.

To construct the "Total Received" amount, the blockchain explorer first will decode the Base58Check encoding of the bitcoin address to retrieve the 160-bit hash of Bob's public key that is encoded within the address. Then, the blockchain explorer will search through the database of transactions, looking for outputs with P2PKH locking scripts that contain Bob's public key hash. By summing up the value of all the outputs, the blockchain explorer can produce the total value received.

Constructing the current balance (displayed as "Final Balance") requires a bit more work. The blockchain explorer keeps a separate database of the outputs that are currently unspent, the UTXO set. To maintain this database, the blockchain explorer must monitor the bitcoin network, add newly created UTXO, and remove spent UTXO, in real time, as they appear in unconfirmed transactions. This is a complicated process that depends on keeping track of transactions as they propagate, as well as maintaining consensus with the bitcoin network to ensure that the correct chain is followed. Sometimes, the blockchain explorer goes out of sync and its perspective of the UTXO set is incomplete or incorrect.

From the UTXO set, the blockchain explorer sums up the value of all unspent outputs referencing Bob's public key hash and produces the "Final Balance" number shown to the user.

In order to produce this one image, with these two "balances," the blockchain explorer has to index and search through dozens, hundreds, or even hundreds of thousands of transactions.

In summary, the information presented to users through wallet applications, blockchain explorers, and other bitcoin user interfaces is often composed of higher-level abstractions that are derived by searching many different transactions, inspecting their content, and manipulating the data contained within them. By presenting this simplistic view of bitcoin transactions that resemble bank checks from one sender to one recipient, these applications have to abstract a lot of underlying detail. They

mostly focus on the common types of transactions: P2PKH with SIGHASH_ALL signatures on every input. Thus, while bitcoin applications can present more than 80% of all transactions in an easy-to-read manner, they are sometimes stumped by transactions that deviate from the norm. Transactions that contain more complex locking scripts, or different SIGHASH flags, or many inputs and outputs, demonstrate the simplicity and weakness of these abstractions.

Every day, hundreds of transactions that do not contain P2PKH outputs are confirmed on the blockchain. The blockchain explorers often present these with red warning messages saying they cannot decode an address. The following link contains the most recent "strange transactions" that were not fully decoded: *https://blockchain.info/strange-transactions*.

As we will see in the next chapter, these are not necessarily strange transactions. They are transactions that contain more complex locking scripts than the common P2PKH. We will learn how to decode and understand more complex scripts and the applications they support next.