Chapter 5 - 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 an public entry in bitcoin's global double-entry bookkeeping ledger, the blockchain.

In this chapter we will examine all the various forms of transactions, what they contain, how to create them, how they are verified, and become part of the permanent record of all transactions.

Transaction Lifecycle

A transaction's lifecycle starts with the transaction's creation, also known as origination. The transaction is then signed, with one or more signatures indicating the authorization to spend the funds referenced by the transaction. The transaction is then broadcast on the bitcoin network, where each network node (participant) validates and propagates the transaction until it reaches (almost) every node in the network. Finally, the transaction is verified by a mining node and included in a block of transactions that is recorded on the blockchain. Once recorded on the blockchain and confirmed by sufficient subsequent blocks (confirmations), the transaction is a permanent part of the bitcoin ledger and is accepted as valid by all participants. The funds allocated to a new owner by the transaction can then be spend in a new transaction, extending the chain of ownership and beginning the lifecycle of a transaction again.

Creating Transactions

In some ways it helps to think of a transaction in the same way as a paper cheque. Like a cheque, a transaction is an instrument that expresses the intent to transfer money and is not visible to the financial system until it is submitted for execution. Like a cheque, the originator of the transaction does not have to be the one signing the transaction. Transactions can be created online or offline by anyone, even if the person creating the transaction is not an authorized signer on the account. For example an accounts payable clerk might process payable cheques for signature by the CEO. Similarly, an accounts payable clerk can create bitcoin transactions and then have the CEO apply digital signatures to make them valid. While a cheque references a specific account as the source of the funds, a bitcoin transaction references a specific previous transaction as its source, rather than an account.

Once a transaction has been created, it is signed by the owner (or owners) of the source funds. If it was properly formed and signed, the signed transaction is now valid and contains all the information needed to execute the transfer of funds. Finally, the valid transaction has to reach the bitcoin network so that it can be propagated until it reaches a miner for inclusion in the pubic ledger, the blockchain.

Broadcasting Transactions to the Bitcoin Network

Next, a transaction needs to be delivered to the bitcoin network so that it can be propagated and be included in the blockchain. In essence, a bitcoin transaction is just 300-400 bytes of data and has to reach any one of tens of thousands of bitcoin nodes. The sender does not need to trust the nodes they use to broadcast the transaction, as long as they use more than one to ensure that it propagates. The nodes don't need to trust the sender or establish the sender's "identity". Since the transaction is signed and contains no confidential information, private keys or credentials, it can be publicly broadcast using any underlying network transport that is convenient. Unlike credit card transactions, for example, which contain sensitive information and can only be transmitted on encrypted networks, a bitcoin transaction can be sent over any network. As long as the transaction can reach a bitcoin node that will propagate it into the bitcoin network, it doesn't matter how it is transported to the first node. Bitcoin transactions can therefore be transmitted to the bitcoin network over insecure networks such as Wifi, Bluetooth, NFC, Chirp, barcodes or by copy and paste in a web form. In extreme cases, a bitcoin transaction could be transmitted over packet radio, satellite relay or shortwave using burst transmission, spread spectrum or frequency hoping to evade detection and jamming. A bitcoin transaction could even be encoded as smileys (emoticons) and posted in a public forum or sent as a text message or Skype chat message. Bitcoin has turned money into a data structure making it virtually impossible to stop anyone from creating and executing a bitcoin transaction.

Propagating Transactions on the Bitcoin Network

Once a bitcoin transaction is sent to any node connected to the bitcoin network, the transaction will be validated by that node. If valid, that node will propagate it to the other

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

nodes it is connected to and a success message will be returned synchronously to the originator. If the transaction is invalid, the node will reject it and synchronously return a rejection message to the originator. The bitcoin network is a peer-to-peer network, meaning that each bitcoin node is connected to a few other bitcoin nodes which it discovers during startup through the peer-to-peer protocol. The entire network forms a loosely connected mesh without a fixed topology or any structure, making all nodes equal peers. Messages, including transactions and blocks are propagated from each node to the peers it is connected to. A new validated transaction injected into any node on the network will be sent to 3-4 of the neighboring nodes, each of which will send it to 3-4 more nodes and so on. In this way, within a few seconds a valid transaction will propagate in an exponentially expanding ripple across the network until all connected nodes have received it. The bitcoin network is designed to propagate transactions and blocks to all nodes in an efficient and resilient manner that is resistant to attacks. To prevent spamming, denial of service attacks or other nuisance attacks agains the bitcoin system, every node will independently validate every transaction before propagating it further. A malformed transaction will not get beyond one node. The rules by which transactions are validated are explained in more detail in [tx_validation]

Mining Transactions into Blocks

Some of the nodes in the bitcoin network participate in "mining". Mining is the process creating new blocks of transactions that will become part of the blockchain. Miners collect transactions and group them into blocks, they then attempt to prove each block with the proof-of-work algorithm. Blocks with a valid proof of work are added to and extend the linked chain of blocks called the blockchain. Once a transaction is added to the blockchain, the new owner of the funds can reference it in a new transaction and spend the funds.

The blockchain forms the authoritative ledger of all transactions since bitcoin's beginning in 2009. The blockchain is the subject of the next chapter, where we will examine the formation of the authoritative record through the competitive process of proof-of-work, also known as mining.

Transaction Structure

A transaction is a data structure that encodes a transfer of value from a source of funds, called an "input", to a destination, called an "output". Transaction inputs and outputs are not related to accounts or identities. Instead you should think of them as bitcoin amounts, chunks of bitcoin, being locked with a specific secret which only the owner, or person know knows the secret, can unlock.

A transaction contains a number of fields, in addition to the inputs and outputs, as follows:

Table 1. The structure of a transaction

Size	Field	Description
4 bytes	Version	Specifies which rules this transaction follows
1-9 bytes (VarInt)	Input Counter	How many inputs are included
Variable	Inputs	One or more Transaction Inputs
1-9 bytes (VarInt)	Output Counter	How many outputs are included
Variable	Outputs	One or more Transaction Outputs
4 bytes	Locktime	A unix timestamp or block number

Note: Locktime defines the earliest time that a transaction can be added to the blockchain. It is set to zero in most transactions to indicate immediate execution. If locktime is non-zero and below 500 million, it is interpreted as a block height, meaning the transaction is not included in the blockchain prior to the specified block height. If it is above 500 million, it is interpreted as a Unix Epoch timestamp (seconds since Jan-1-1970) and the transaction is not included in the blockchain prior to the specified time.

Transaction Outputs and Inputs

The fundamental building block of a bitcoin transaction is an *unspent transaction output* or UTXO. UTXO are indivisible chunks of bitcoin currency locked to a specific

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

owner, recorded on the blockchain, and recognized as currency units by the entire network. The bitcoin network tracks all available (unspent) UTXO currently numbering in the millions. Whenever a user receives bitcoin, that amount is recorded within the blockchain as a UTXO. Thus, a user's bitcoin may be scattered as UTXO amongst hundreds of transactions and hundreds of blocks. In effect, there is no such thing as a balance of a bitcoin address or account; there are only scattered UTXO, locked to specific owners. The concept of a user's bitcoin balance is a construct created by the wallet application. The wallet calculates the user's balance by scanning the blockchain and aggregating all UTXO belonging to that user.

Tip There are no accounts or balances in bitcoin, there are only *unspent transaction outputs* (UTXO) scattered in the blockchain.

Unlike cash which exists in specific denominations, one dollar, five dollars, ten dollars, etc., a UTXO can have any arbitrary value denominated as a multiple of satoshis (the smallest bitcoin unit equal to 100 millionth of a bitcoin). While UTXO can be any arbitrary value, once created it is indivisible just like a coin that cannot be cut in half. If a 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 20 bitcoin UTXO and want to pay 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, bitcoin transactions must occasionally generate change.

In simple terms, transactions consume the sender's available UTXO and create new UTXO locked to the recipient's bitcoin address. Imagine a shopper buying a $1.50 beverage, reaching into their 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 (a dollar bill and two quarters), or a combination of smaller denominations (six quarters), or if necessary, a larger unit such as a bank note (five dollar note). If they hand too much money, say $5, to the shop owner they will expect $3.50 change, which they will return to their wallet and have available for future transactions. Similarly, a bitcoin transaction must be created from a users UTXO in whatever denominations that user has available. They cannot cut a UTXO in half anymore than they can cut a dollar bill in half and use it as currency. The user's wallet application will typically select from the users available UTXO various units 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.

The UTXO consumed by a transaction are called transaction inputs, while the UTXO created by a transaction are called transaction outputs. This way, chunks of bitcoin value move forward from owner to owner in a chain of transactions consuming and creating UTXO. Transactions consume UTXO unlocking it with the signature of the current owner and create UTXO locking it to the bitcoin address of the new owner.

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 is how bitcoin's money supply is created during the mining process as we will see in [mining]

Tip 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 [op_return]) create spendable chunks of bitcoin called *unspent transaction outputs* or UTXO, which are then recognized by the whole network and available for the owner to spend in a future transaction. Sending someone bitcoin is creating an unspent transaction output (UTXO) registered to their address and available for them to spend.

UTXO are tracked by every full node bitcoin client in a database held in memory, called the *UTXO set* or *UTXO pool*. 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 *locking script*, also known as an "encumbrance" that "locks" this amount by specifying the conditions that must be met to spend the output

The transaction scripting language, used in the locking script mentioned above, is discussed in detail in [tx_script]

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

Table 2. The structure of a transaction output

Size	Field	Description
8 bytes	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

Spending Conditions (Encumbrances)

Transaction outputs associate a specific amount (in satoshis) to a specific *encumbrance* or locking-script that defines the condition that must be met to spend that amount. In most cases the locking script will lock the output to a specific bitcoin address, thereby transferring ownership of that amount to the new owner. When Alice paid Bob's Cafe for a cup of coffee, her transaction created a 0.015 bitcoin output *encumbered* or locked to the Cafe's bitcoin address. That 0.015 bitcoin output was recorded on the blockchain and became part of the Unspent Transaction Output set, meaning it showed in Bob's wallet as part of the available balance. When Bob chooses to spend that amount, his transaction will release the encumbrance, unlocking the output by providing an unlocking script containing a signature from Bob's private key.

Transaction Inputs

In simple terms, transaction inputs are pointers to UTXO. They point to a specific UTXO by reference to the transaction hash and sequence number where the UTXO is recorded in the blockchain. To spend UTXO, a transaction input also includes unlocking-scripts that satisfy the spending conditions set by the UTXO. The unlocking script is usually a signature proving ownership of the bitcoin address that is in the locking script.

When a user makes a payment, their wallet constructs a transaction by selecting from the available UTXO. For example, to make a 0.015 bitcoin payment, the wallet app may select a 0.01 UTXO and a 0.005 UTXO, using them both to add up to the desired payment amount. The wallet then produces unlocking scripts containing signatures for each of the UTXO, thereby making them spendable by satisfying their locking script conditions. The wallet adds these UTXO references and unlocking scripts as inputs to the transaction.

Table 3. The structure of a transaction input

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	Currently-disabled Tx-replacement feature, set to 0xFFFFFFFF

Note: The sequence number is used to override a transaction prior to the expiration of the transaction locktime, which is a feature that is currently disabled in bitcoin. Most transactions set this value to the maximum integer value (0xFFFFFFFF) and it is ignored by the bitcoin network. If the transaction has a non-zero locktime, at least one of

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

its inputs must have a sequence number below 0xFFFFFFFF in order to enable locktime.

Transaction Fees

Most transactions include transactions fees that compensate the bitcoin miners for securing the network. Mining and the fees and rewards collected by miners are discussed in more detail in [mining]. 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 fees.

Transaction fees serve as an incentive to include (mine) a transaction into the next block and also as a disincentive against "spam" transactions or any kind of 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 may 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-most mined block, while a transaction with insufficient or no fees may be delayed, on a best-effort basis and processed after a few blocks or not at all. Transaction fees are not mandatory and transactions without fees may be processed, eventually, but including transaction fees encourages priority processing.

Over time, the way transaction fees are calculated and the effect they have on transaction prioritization has been changing. At first, transaction fees were fixed and constant across the network. Gradually, the fee structure has been relaxed so that it may be influenced by market forces, based on network capacity and transaction volume. The current minimum transaction fee is fixed at 0.0001 bitcoin or a tenth of a milli-bitcoin, recently decreased from one milli-bitcoin, per kilobyte. Most transactions are less than one kilobyte, however those with multiple inputs or outputs can be larger. In future revisions of the bitcoin protocol it is expected that wallet applications will use statistical analysis to calculate the most appropriate fee to attach to a transaction based on the average fees of recent transactions.

The current algorithm used by miners to prioritize transactions for inclusion in a block based on their fees will be examined in detail in [mining].

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.

Transaction fees are implied, as the excess of inputs minus outputs

```
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!

> **Warning** 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!" may not be what you intended.

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. While you will receive priority processing and make a miner very happy, this is probably not 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 school books for the children. She

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

received several thousand small donations from people all around the world, totaling 50 . Now, she wants to purchase hundreds of school books from a local publisher, paying in bitcoin. The charity received thousands of small donations from all around the world. 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 2-3 kilobytes in size. As a result, it will require a higher fee than the minimal network fee of 0.0001 bitcoin. 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

Bitcoin clients validate transactions by executing a script, written in a Forth-like scripting language. Both the locking script (encumbrance) placed on a UTXO and the unlocking script that usually contains a signature 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.

Today most transactions processed through the bitcoin network have the form "Alice pays Bob" and are based on the same script called a Pay-to-Public-Key-Hash script. However, the use of scripts to lock outputs and unlock inputs means that through use of the programming language, transactions can contain an infinite number of conditions. Bitcoin transactions are not limited to the "Alice pays Bob" form and pattern.

This is only the tip of the iceberg of possibilities that can be expressed with this scripting language. In this section we will demonstrate the components of bitcoins transaction scripting language and show how it can be used to express complex conditions for spending and how those conditions can be satisfied by unlocking scripts.

> **Tip** 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 near infinite variety of conditions to be expressed. This is how bitcoin gets the power of "programmable money"

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 an encumbrance placed on an output, that 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. 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".

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 and most of the time they contain a digital signature produced by the user's wallet from their private key. Historically, the unlocking script is called *scriptSig*, because it usually contained a digital signature. In this book we refer to it as an "unlocking script" to acknowledge the much broader range of locking script requirements, as not all unlocking scripts must contain signatures. As mentioned above, in most bitcoin applications the source code will refer to the unlocking script as "scriptSig".

Every bitcoin client will validate transaction by executing the locking and unlocking scripts together. For each input in the transaction, the validation software will first retrieve the UTXO referenced by the input. That UTXO contains a locking script defining the conditions required to spend it. The validation software will then take the unlocking script contained in the input that is attempting to spend this UTXO and concatenate them. The locking script is added to the end of the unlocking script and then the entire combined script is executed using the script execution engine. If the result of executing the combined 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 as it has failed to satisfy the spending conditions placed on the UTXO. 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 UTXO results in the UTXO being marked as "spent" and removed from the set of available UTXO.

Below 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:

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

Figure 1. Combining scriptSig and scriptPubKey to evaluate a transaction script

Scripting Language

The bitcoin transaction script language, also named confusingly *Script*, is a Forth-like reverse-polish notation stack-based execution language. If that sounds like gibberish, you probably haven't studied 1960's programming languages. Script is a very simple, lightweight language that was designed to be limited in scope and executable on a range of hardware, perhaps as simple as an embedded device, like a handheld calculator. It requires minimal processing and cannot do many of the fancy things modern programming languages can do. In the case of programmable money, that is a deliberate security feature.

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, which 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.

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 may 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 result TRUE that signifies a valid transaction.

In the following example, 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 the resulting sum is equal to `5`. For brevity, the OP_ prefix is omitted in the step-by-step example.

Figure 2. Bitcoin's script validation doing simple math

Below is a slightly more complex script, which calculates `((2 + 3) * 2) + 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 3 OP_ADD 2 OP_MUL 1 OP_ADD 11 OP_EQUAL
```

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

While most locking scripts refer to a bitcoin address or public key, 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 above 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 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
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

As we saw in the step-by-step example above, 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.

> **Tip** Transactions are valid if the top result on the stack is TRUE (1), any other non-zero value or if the stack is empty after script execution. Transactions are invalid if the top value on the stack is FALSE (0) 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 [tx_script_ops] for details.

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. 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 a key benefit of the bitcoin system.

Standard Transactions

In the first few years of bitcoin's development, the developers introduced some limitations in the types of scripts that could be processed by the reference client. These limitations are encoded in a function called `isStandard()` which defines five types of "standard" transactions. These limitations are temporary and may be lifted by the time you read this. Until then, the five standard types of transaction scripts are the only ones that will be accepted by the reference client and most miners who run the reference client. While it is possible to create a non-standard transaction containing a script that is not one of the standard types, you must find a miner who does not follow these limitations, to mine that transaction into a block.

Check the source code of the bitcoin core client (the reference implementation) to see what is currently allowed as a valid transaction script.

The five standard types of transaction scripts are Pay-to-Public-Key-Hash (P2PKH), Public-Key, Multi-Signature (limited to 15 keys), Pay-to-Script-Hash (P2SH) and Data Output (OP_RETURN), which are described in more detail below.

Pay to Public Key Hash (P2PKH)

The vast majority of transactions processed on the bitcoin network are Pay-to-Public-Key-Hash, also known as P2PKH transactions. These contain a locking script that encumbers the output with a public key hash, more commonly known as a bitcoin address. Transactions that pay a bitcoin address contain P2PKH scripts. 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.

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_EQUAL 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 locking script above can be satisfied with an unlocking script of the form:

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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

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

```
<Cafe Signature> <Cafe Public Key> OP_DUP OP_HASH160 <Cafe Public Key Hash> OP_EQUAL 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 which corresponds to the public key hash set as an encumbrance.

Here's a step-by-step execution of the combined script, which will prove this is a valid transaction:

Figure 3. Evaluating a script for a Pay-to-Public-Key-Hash transaction (Part 1 of 2)

Figure 4. Evaluating a script for a Pay-to-Public-Key-Hash transaction (Part 2 of 2)

Pay-to-Public-Key

Pay-to-Public-Key is a simpler form of a bitcoin payment than Pay-to-Public-Key-Hash. With this script form, the public key itself is stored in the locking script, rather than a public-key-hash as with P2PKH above, which is much shorter. The disadvantage of this form of locking script is that it consumes more space in the blockchain to store these types of payments, because a public key is 264 or 520 bits long (depending on whether it is compressed), whereas a public key hash is only 160 bits long. For legacy compatibility, Pay-to-Public-Key is used in all coinbase generation transactions, the transactions that pay the reward to the miners.

A Pay-to-Public-Key locking script looks like this:

```
<Public Key A> OP_CHECKSIG
```

The corresponding unlocking script that must be presented to unlock this type of output is a simple signature, like this:

```
<Signature from Private Key A>
```

The combined script, which is validated by the transaction validation software is:

```
<Signature from Private Key A> <Public Key A> OP_CHECKSIG
```

The script above is a simple invocation of the CHECKSIG operator which validates the signature as belonging to the correct key and returns TRUE on the stack.

Mutli-Signature

Multi-signature scripts set a condition where N public keys are recorded in the script and at least M of those must provide signatures to release the encumbrance. This is also known as an M-of-N scheme, where N is the total number of keys and M is the threshold of signatures required for validation. For example, a 2-of-3 multi-signature is one where 3 public keys are listed as potential signers and at least 2 of those must be used to create signatures for a valid transaction to spend the funds. At this time, standard multi-signature scripts are limited to at most 15 listed public keys, meaning you can do anything from a 1-of-1 to a 15-of-15 multi-signature or any combination within that range. The limitation to 15 listed keys may be lifted by the time of publication of this book, so check the `isStandard()` function to see what is currently accepted by the network.

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

The general form of a locking script setting an M-of-N multi-signature condition is:

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG
```

where N is the total number of listed public keys and M is the threshold of required signatures to spend the output.

A locking script setting a 2-of-3 multi-signature condition looks like this:

```
2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG
```

The locking script above can be satisfied with an unlocking script containing pairs of signatures and public keys:

```
OP_0 <Signature B> <Signature C>
```

or any combination of two signatures from the private keys corresponding to the three listed public keys.

Note: The prefix OP_0 is required because of a bug in the original implementation of CHECKMULTISIG where one item too many is popped off the stack. It is ignored by CHECKMULTISIG and is simply a placeholder.

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

```
OP_0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG
```

When executed, this combined script will evaluate to TRUE if, and only if, the unlocking script matches the conditions set by the locking script, that is if the unlocking script has a valid signatures from the two private keys which correspond to two of the three public keys set as an encumbrance.

Data Output (OP_RETURN)

Bitcoin's distributed and timestamped ledger, the blockchain, has potential uses far beyond payments. Many developers have tried to use the transaction scripting language to take advantage of the security and resilience of the system for applications such as digital notary services, stock certificates, and smart contracts. Early attempts to use bitcoin's script language for these purposes involved creating transaction outputs that recorded data on the blockchain, for example to record a digital fingerprint of a file in such a way that anyone could establish proof-of-existence of that file on a specific date by reference to that transaction.

The use of bitcoin's blockchain to store data unrelated to bitcoin payments is a controversial subject. Many developers consider such use abusive and want to discourage it. Others view it as a demonstration of the powerful capabilities of blockchain technology and want to encourage such experimentation. Those who object to the inclusion of non-payment data argue that it causes "blockchain bloat", burdening those running full bitcoin nodes with carrying the cost of disk storage for data that the blockchain was not intended to carry. Moreover, such transactions create UTXO that cannot be spent, using the destination bitcoin address as a free-form 20-byte field. Since the address is used for data, it doesn't correspond to a private key and the resulting UTXO can *never* be spent, it's a fake payment. This practice causes the size of the in-memory UTXO set to increase and these transactions which can never be spent are therefore never removed, forcing bitcoin nodes to carry these forever in RAM which is far more expensive.

In version 0.9 of the bitcoin core client, a compromise was reached, with the introduction of the OP_RETURN operator. OP_RETURN allows developers to add 40 bytes of non-payment data to a transaction output. However, unlike the use of "fake" UTXO, the OP_RETURN operator creates an explicitly *provably un-spendable* output, which does not need to be stored in the UTXO set. OP_RETURN outputs are recorded on the blockchain, so they consume disk space and contribute to the increase in the blockchain's size, but they are not stored in the UTXO set and therefore do not bloat the UTXO memory pool and burden full nodes with the cost of more expensive RAM.

OP_RETURN scripts look like this:

```
OP_RETURN <data>
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

where the data portion is limited to 40 bytes and most often represents a hash, such as the output from the SHA256 algorithm (32 bytes). Many applications put a prefix in front of the data to help identify the application. For example, the proofofexistence.com digital notarization service uses the 8-byte prefix "DOCPROOF" which is ASCII encoded as 44f4350524f4f46 in hexadecimal.

Keep in mind that there is no "unlocking script" that corresponds to OP_RETURN, that can be used to "spend" an OP_RETURN output. The whole point of OP_RETURN is that you can't spend the money locked in that output and therefore it does not need to be held in the UTXO set as potentially spendable - OP_RETURN is *provably unspendable*. OP_RETURN is usually an output with a zero bitcoin amount, since any bitcoin assigned to such an output is effectively lost forever. If an OP_RETURN is encountered by the script validation software it results immediately in halting the execution of the validation script and marking the transaction as invalid. Thus, if you accidentally reference an OP_RETURN output as an input in a transaction, that transaction is invalid.

A valid transaction can have only one OP_RETURN output. However, a single OP_RETURN output can be combined in a transaction with outputs of any other type.

Pay to Script Hash (P2SH)

Pay-to-Script-Hash (P2SH) was introduced in the winter of 2012 as a powerful new type of transaction that greatly simplifies the use of complex transaction scripts. To explain the need for P2SH, let's look at a practical example.

In chapter 1 we introduced Mohammed, an electronics importer based in Dubai. Mohammed's company uses bitcoin's multi-signature feature extensively for its corporate accounts. Multi-signature scripts are one of the most common uses of bitcoin's advanced scripting capabilities and are a very powerful feature. Mohammed's company uses a multi-signature script for all customer payments, known in accounting terms as "accounts receivable" or AR. With the multi-signature scheme, any payments made by customers are locked in such a way that they require at least two signatures to release, from Mohammed and one of his partners or from his attorney who has a backup key. A multi-signature scheme like that offers corporate governance controls and protects against theft, embezzlement or loss.

The resulting script is quite long and looks like this:

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5
OP_CHECKMULTISIG
```

While multi-signature scripts are a powerful feature, they are cumbersome to use. Given the script above, Mohammed would have to communicate this script to every customer prior to payment. Each customer would have to use special bitcoin wallet software with the ability to create custom transaction scripts and each customer would have to understand how to create a transaction using custom scripts. Furthermore, the resulting transaction would be about five times larger than a simple payment transaction, as this script contains very long public keys. The burden of that extra-large transaction would be borne by the customer in the form of fees. Finally, a large transaction script like this would be carried in the UTXO set in RAM in every full node, until it was spent. All of these issues make using complex output scripts difficult in practice.

Pay-to-Script-Hash (P2SH) was developed to resolve these practical difficulties and to make the use of complex scripts as easy as a payment to a bitcoin address. With P2SH payments, the complex locking script is replaced with its digital fingerprint, a cryptographic hash. When a transaction attempting to spend the UTXO is presented later, it must contain the script that matches the hash, in addition to the unlocking script. In simple terms, P2SH means "pay to a script matching this hash, a script which will be presented later when this output is spent".

In P2SH transactions, the locking script that is replaced by a hash is referred to as the *redeemScript* because it is presented to the system at redemption time rather than as a locking script.

Table 4. Complex Script Without P2SH

Locking Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
Unlocking Script	Sig1 Sig2

Table 5. Complex Script as P2SH

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

Redeem Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
Locking Script	OP_HASH160 <20-byte hash of redeemScript> OP_EQUAL
Unlocking Script	Sig1 Sig2 redeemScript

As you can see from the tables above, with P2SH the complex script that details the conditions for spending the output (redeemScript) is not presented in the locking script. Instead, only a hash of it is in the locking script and the redeemScript itself is presented later, as part of the unlocking script when the output is spent. This shifts the burden in fees and complexity from the sender to the recipient (spender) of the transaction.

Let's look at Mohammed's company, their complex multi-signature script and the resulting P2SH scripts.

First, the multi-signature script that Mohammed's company uses for all incoming payments from customers:

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5
OP_CHECKMULTISIG
```

If the placeholders above are replaced by actual public keys (shown below as 520 bit numbers starting with 04) you can see that this script becomes very long:

```
2
04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C587

04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49

047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D9977965

0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5

043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800

5 OP_CHECKMULTISIG
```

The entire script above can instead be represented by a 20-byte cryptographic hash, by first applying the SHA256 hashing algorithm and then applying the RIPEMD160 algorithm on the result. The 20-byte hash of the above script is:

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

A P2SH transaction locks the output to this hash instead of the longer script, using the locking script:

```
OP_HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e OP_EQUAL
```

which, as you can see is much much shorter. Instead of "pay to this 5-key multi-signature script", the P2SH equivalent transaction is "pay to a script with this hash". A customer making a payment to Mohammed's company need only include this much shorter locking script in their payment. When Mohammed wants to spend this UTXO, they must present the original redeemScript (the one whose hash locked the UTXO) and the signatures necessary to unlock it, like this:

```
<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG>
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

The two scripts are combined in two stages. First, the redeemScript is checked against the locking script to make sure the hash matches:

```
<2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG> OP_HASH160 <redeemScriptHash> OP_EQUAL
```

If the redeemScript hash matches, then the unlocking script is executed on its own, to unlock the redeemScript:

```
<Sig1> <Sig2> 2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG
```

Pay-to-Script-Hash Addresses

Another important part of the P2SH feature is the ability to encode a script hash as an address, as defined in BIP0013. P2SH addresses are Base58Check encodings of the 20-byte hash of a script, just like bitcoin addresses are Base58Check encodings of the 20-byte hash of a public key. P2SH addresses use the version prefix "5", which results in Base58Check encoded addresses that start with a "3". For example, Mohammed's complex script, hashed and Base58Check encoded as P2SH address becomes `39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw`. Now, Mohammed can give this "address" to his customers and they can use almost any bitcoin wallet to make a simple payment, as if it were a bitcoin address. The 3 prefix gives them a hint that this is a special type of address, one corresponding to a script instead of a public key, but otherwise it works in exactly the same way as a payment to a bitcoin address.

P2SH addresses hide all of the complexity, so that the person making a payment does not see the script.

Benefits of Pay-to-Script-Hash

The Pay-to-Script-Hash feature offers the following benefits compared to the direct use of complex scripts in locking outputs:

- Complex scripts are replaced by shorter fingerprint in the transaction output, making the transaction smaller
- Scripts can be coded as an address, so the sender and the sender's wallet don't need complex engineering to implement P2SH
- P2SH shifts the burden of constructing the script to the recipient not the sender
- P2SH shifts the burden in data storage for the long script from the output (which is in UTXO set) to the input (only stored on the blockchain)
- P2SH shifts the burden in data storage for the long script from the present time (payment) to a future time (when it is spent)
- P2SH shifts the transaction fee cost of a long script from the sender to the recipient who has to include the long redeemScript to spend it

Redeem Script and isStandard Validation

Pay-to-Script-Hash is currently limited to the standard types of bitcoin transaction scripts, by the `isStandard()` function. That means that the redeemScript presented in the spending transaction must be one of the standard types: P2PK, P2PKH or Multi-Sig, excluding OP_RETURN and P2SH itself. You cannot reference a P2SH script inside a redeemScript and you can't use an OP_RETURN inside a P2SH redeemScript.

This limitation of redeemScript to only standard transaction scripts is temporary and will likely be removed in future versions of the bitcoin reference implementation, allowing the use of any valid script inside a P2SH redeemScript. You will still not be able to put a P2SH inside a P2SH redeemScript, because the P2SH specification is not recursive. You will still not be able to use OP_RETURN in a redeemScript because OP_RETURN cannot be redeemed by definition. But you will be able to use all the other operators to create a vast range of complex and novel scripts that can be used as redeemScripts and referenced as P2SH payment to their hash.

Note that since the redeemScript is not presented to the network until you attempt to spend a P2SH output, if you lock an output with the hash of a non-standard transaction it will be processed as valid. However, you will not be able to spend it as the spending transaction which includes the redeemScript will not be accepted, as it is non-standard. This creates a risk, as you can lock bitcoin in a P2SH which cannot be later spent. The network will accept the P2SH encumbrance even if it corresponds to a non-standard or invalid redeemScript, because the script hash gives no indication of the script it represents.

Warning P2SH locking scripts contain the hash of a redeemScript which gives no clues as to the content of the redeemScript itself. The P2SH

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

> transaction will be considered valid and accepted even if the redeemScript is invalid or non-standard. You may accidentally lock bitcoin in such a way that it cannot later be spent.

Transaction Script Language Operators, Constants and Symbols

Table 6. Push Value onto Stack

Symbol	Value (hex)	Description
OP_0 or OP_FALSE	0x00	An empty array is pushed on to the stack
1-75	0x01-0x4b	Push the next N bytes onto the stack, where N is 1 to 75 bytes
OP_PUSHDATA1	0x4c	The next script byte contains N, push the following N bytes onto the stack
OP_PUSHDATA2	0x4d	The next two script bytes contain N, push the following N bytes onto the stack
OP_PUSHDATA4	0x4e	The next four script bytes contain N, push the following N bytes onto the stack
OP_1NEGATE	0x4f	Push the value "-1" onto the stack
OP_RESERVED	0x50	Halt - Invalid transaction unless found in an unexecuted OP_IF clause
OP_1 or OP_TRUE	0x51	Push the value "1" onto the stack
OP_2 to OP_16	0x52 to 0x60	For OP_N, push the value "N" onto the stack. eg. OP_2 pushes "2"

Table 7. Conditional Flow Control

Symbol	Value (hex)	Description
OP_NOP	0x61	Do nothing
OP_VER	0x62	Halt - Invalid transaction unless found in an unexecuted OP_IF clause
OP_IF	0x63	Execute the statements following if top of stack is not 0
OP_NOTIF	0x64	Execute the statements following if top of stack is 0
OP_VERIF	0x65	Halt - Invalid transaction

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

OP_VERNOTIF	0x66	Halt - Invalid transaction
OP_ELSE	0x67	Execute only if the previous statements were not executed
OP_ENDIF	0x68	Ends the OP_IF, OP_NOTIF, OP_ELSE block
OP_VERIFY	0x69	Check the top of the stack, Halt and Invalidate transaction if not TRUE
OP_RETURN	0x6a	Halt and invalidate transaction

Table 8. Stack Operations

Symbol	Value (hex)	Description
OP_TOALTSTACK	0x6b	Pop top item from stack and push to alternative stack
OP_FROMALTSTACK	0x6c	Pop top item from alternative stack and push to stack
OP_2DROP	0x6d	Pop top two stack items
OP_2DUP	0x6e	Duplicate top two stack items
OP_3DUP	0x6f	Duplicate top three stack items
OP_2OVER	0x70	Copies the third and fourth items in the stack to the top
OP_2ROT	0x71	Moves the fifth and sixth items in the stack to the top
OP_2SWAP	0x72	Swap the two top pairs of items in the stack
OP_IFDUP	0x73	Duplicate the top item in the stack if it is not 0
OP_DEPTH	0x74	Count the items on the stack and push the resulting count
OP_DROP	0x75	Pop the top item in the stack
OP_DUP	0x76	Duplicate the top item in the stack
OP_NIP	0x77	Pop the second item in the stack
OP_OVER	0x78	Copy the second item in the stack and push it on to the top
OP_PICK	0x79	Pop value N from top, then copy the Nth item to the top of the stack
OP_ROLL	0x7a	Pop value N from top, then move the Nth item to the top of the stack

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

OP_ROT	0x7b	Rotate the top three items in the stack
OP_SWAP	0x7c	Swap the top three items in the stack
OP_TUCK	0x7d	Copy the top item and insert it between the top and second item.

Table 9. String Splice Operations

Symbol	Value (hex)	Description
OP_CAT	0x7e	Disabled (Concatenates top two items)
OP_SUBSTR	0x7f	Disabled (Returns substring)
OP_LEFT	0x80	Disabled (Returns left substring)
OP_RIGHT	0x81	Disabled (Returns right substring)
OP_SIZE	0x82	Calculate string length of top item and push the result

Table 10. Binary Arithmetic and Conditionals

Symbol	Value (hex)	Description
OP_INVERT	0x83	Disabled (Flip the bits of the top item)
OP_AND	0x84	Disabled (Boolean AND of two top items)
OP_OR	0x85	Disabled (Boolean OR of two top items)
OP_XOR	0x86	Disabled (Boolean XOR of two top items)
OP_EQUAL	0x87	Push TRUE (1) if top two items are exactly equal, push FALSE (0) otherwise
OP_EQUALVERIFY	0x88	Same as OP_EQUAL, but run OP_VERIFY after to halt if not TRUE
OP_RESERVED1	0x89	Halt - Invalid transaction unless found in an unexecuted OP_IF clause
OP_RESERVED2	0x8a	Halt - Invalid transaction unless found in an unexecuted OP_IF clause

Table 11. Numeric Operators

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

Symbol	Value (hex)	Description
OP_1ADD	0x8b	Add 1 to the top item
OP_1SUB	0x8c	Subtract 1 from the top item
OP_2MUL	0x8d	Disabled (Multiply top item by 2)
OP_2DIV	0x8e	Disabled (Divide top item by 2)
OP_NEGATE	0x8f	Flip the sign of top item
OP_ABS	0x90	Change the sign of the top item to positive
OP_NOT	0x91	If top item is 0 or 1 boolean flip it, otherwise return 0
OP_0NOTEQUAL	0x92	If top item is 0 return 0, otherwise return 1
OP_ADD	0x93	Pop top two items, add them and push result
OP_SUB	0x94	Pop top two items, subtract first form second, push result
OP_MUL	0x95	Disabled (Multiply top two items)
OP_DIV	0x96	Disabled (Divide second item by first item)
OP_MOD	0x97	Disabled (Remainder divide second item by first item)
OP_LSHIFT	0x98	Disabled (Shift second item left by first item number of bits)
OP_RSHIFT	0x99	Disabled (Shift second item right by first item number of bits)
OP_BOOLAND	0x9a	Boolean AND of top two items
OP_BOOLOR	0x9b	Boolean OR of top two items
OP_NUMEQUAL	0x9c	Return TRUE if top two items are equal numbers
OP_NUMEQUALVERIFY	0x9d	Same as NUMEQUAL, then OP_VERIFY to halt if not TRUE
OP_NUMNOTEQUAL	0x9e	Return TRUE if top two items are not equal numbers
OP_LESSTHAN	0x9f	Return TRUE if second item is less than top item
OP_GREATERTHAN	0xa0	Return TRUE if second item is greater than top item
OP_LESSTHANOREQUAL	0xa1	Return TRUE if second item is less than or equal to top item

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

OP_GREATERTHANOREQUAL	0xa2	Return TRUE if second item is great than or equal to top item
OP_MIN	0xa3	Return the smaller of the two top items
OP_MAX	0xa4	Return the larger of the two top items
OP_WITHIN	0xa5	Return TRUE if the third item is between the second item (or equal) and first item

Table 12. Cryptographic and Hashing Operations

Symbol	Value (hex)	Description
OP_RIPEMD160	0xa6	Return RIPEMD160 hash of top item
OP_SHA1	0xa7	Return SHA1 hash of top item
OP_SHA256	0xa8	Return SHA256 hash of top item
OP_HASH160	0xa9	Return RIPEMD160(SHA256(x)) hash of top item
OP_HASH256	0xaa	Return SHA256(SHA256(x)) hash of top item
OP_CODESEPARATOR	0xab	Mark the beginning of signature-checked data
OP_CHECKSIG	0xac	Pop a public key and signature and validate the signature for the transaction's hashed data, return TRUE if matching
OP_CHECKSIGVERIFY	0xad	Same as CHECKSIG, then OP_VEIRFY to halt if not TRUE
OP_CHECKMULTISIG	0xae	Run CHECKSIG for each pair of signature and public key provided. All must match. Bug in implementation pops an extra value, prefix with OP_NOP as workaround
OP_CHECKMULTISIGVERIFY	0xaf	Same as CHECKMULTISIG, then OP_VERIFY to halt if not TRUE

Table 13. Non-Operators

Symbol	Value (hex)	Description
OP_NOP1-OP_NOP10	0xb0-0xb9	Does nothing, ignored.

Table 14. Reserved OP codes for internal use by the parser

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

Symbol	Value (hex)	Description
OP_SMALLDATA	0xf9	Represents small data field
OP_SMALLINTEGER	0xfa	Represents small integer data field
OP_PUBKEYS	0xfb	Represents public key fields
OP_PUBKEYHASH	0xfd	Represents a public key hash field
OP_PUBKEY	0xfe	Represents a public key field
OP_INVALIDOPCODE	0xff	Represents any OP code not currently assigned

Last updated 2014-06-25 19:31:48 CDT

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com