CHAPTER 7

Advanced Transactions and Scripting

Introduction

In the previous chapter, we introduced the basic elements of bitcoin transactions and looked at the most common type of transaction script, the P2PKH script. In this chapter we will look at more advanced scripting and how we can use it to build transactions with complex conditions.

First, we will look at *multisignature* scripts. Next, we will examine the second most common transaction script, *Pay-to-Script-Hash*, which opens up a whole world of complex scripts. Then, we will examine new script operators that add a time dimension to bitcoin, through *timelocks*.

Multisignature

Multisignature scripts set a condition where N public keys are recorded in the script and at least M of those must provide signatures to unlock the funds. 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 multisignature is one where three public keys are listed as potential signers and at least two of those must be used to create signatures for a valid transaction to spend the funds. At this time, standard multisignature 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 multisignature or any combination within that range. The limitation to 15 listed keys might be lifted by the time this book is published, so check the `isStandard()` function to see what is currently accepted by the network.

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

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N 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 multisignature condition looks like this:

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

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

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

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

The two scripts together would form the combined validation script:

```
<Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3
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. In this case, the condition is whether the unlocking script has a valid signature from the two private keys that correspond to two of the three public keys set as an encumbrance.

A bug in CHECKMULTISIG execution

There is a bug in `CHECKMULTISIG`'s execution that requires a slight workaround. When `CHECKMULTISIG` executes, it should consume M+N+2 items on the stack as parameters. However, due to the bug, `CHECKMULTISIG` will pop an extra value or one value more than expected.

Let's look at this in greater detail using the previous validation example:

```
<Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3
CHECKMULTISIG
```

First, `CHECKMULTISIG` pops the top item, which is `N` (in this example "3"). Then it pops `N` items, which are the public keys that can sign. In this example, public keys A, B, and C. Then, it pops one item, which is `M`, the quorum (how many signatures are needed). Here M = 2. At this point, `CHECKMULTISIG` should pop the final `M` items, which are the signatures, and see if they are valid. However, unfortunately, a bug in the implementation causes `CHECKMULTISIG` to pop one more item (M+1 total) than it should. The extra item is disregarded when checking the signatures so it has no direct effect on `CHECKMULTISIG` itself. However, an extra value must be present because if it is not present, when `CHECKMULTISIG` attempts to pop on an empty stack, it will cause a stack error and script failure (marking the transaction as invalid). Because the extra item is disregarded it can be anything, but customarily `0` is used.

Because this bug became part of the consensus rules, it must now be replicated forever. Therefore the correct script validation would look like this:

```
0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3
CHECKMULTISIG
```

Thus the unlocking script actually used in multisig is not:

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

but instead it is:

```
0 <Signature B> <Signature C>
```

From now on, if you see a multisig unlocking script, you should expect to see an extra `0` in the beginning, whose only purpose is as a workaround to a bug that accidentally became a consensus rule.

Pay-to-Script-Hash (P2SH)

Pay-to-Script-Hash (P2SH) was introduced in 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 multisignature feature extensively for its corporate accounts. Multisignature 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 multisignature script for all customer payments, known in accounting terms as "accounts receivable," or AR. With the multisignature 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 multisignature 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 CHECKMULTISIG
```

Although multisignature scripts are a powerful feature, they are cumbersome to use. Given the preceding script, 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, because 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 locking scripts difficult in practice.

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 that 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 *redeem script* because it is presented to the system at redemption time rather than as a locking script. Table 7-1 shows the script without P2SH and Table 7-2 shows the same script encoded with P2SH.

Table 7-1. Complex script without P2SH

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

Table 7-2. Complex script as P2SH

Redeem Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
Locking Script	HASH160 <20-byte hash of redeem script> EQUAL
Unlocking Script	Sig1 Sig2 <redeem script>

As you can see from the tables, with P2SH the complex script that details the conditions for spending the output (redeem script) is not presented in the locking script. Instead, only a hash of it is in the locking script and the redeem script 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, the complex multisignature script, and the resulting P2SH scripts.

First, the multisignature 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 CHECKMULTISIG
```

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

```
2
04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395
D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C58704A2192968D8655D6A935BEAF2CA23
E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D7
8D0E34224858008E8B49047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5
737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D99779650421D65CBD
7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D0872274
40645ABE8A35DA8C5B73997AD343BE5C2AFD94A5043752580AFA1ECED3C68D446BCAB69AC0BA7DF5
0D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7E
D238F4D800 5 CHECKMULTISIG
```

This entire script 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 preceding script is:

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

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

```
HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL
```

which, as you can see, is much shorter. Instead of "pay to this 5-key multisignature 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 his payment. When Mohammed and his partners want to spend this UTXO, they must present the original redeem script (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 CHECKMULTISIG>
```

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

```
<2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG> HASH160 <redeem scriptHash> EQUAL
```

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

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

Almost all the scripts described in this chapter can only be implemented as P2SH scripts. They cannot be used directly in the locking script of a UTXO.

P2SH Addresses

Another important part of the P2SH feature is the ability to encode a script hash as an address, as defined in BIP-13. 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 a 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 P2SH

The P2SH feature offers the following benefits compared to the direct use of complex scripts in locking outputs:

- Complex scripts are replaced by shorter fingerprints 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 the UTXO set) to the input (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 redeem script to spend it.

Redeem Script and Validation

Prior to version 0.9.2 of the Bitcoin Core client, Pay-to-Script-Hash was limited to the standard types of bitcoin transaction scripts, by the `isStandard()` function. That means that the redeem script presented in the spending transaction could only be one of the standard types: P2PK, P2PKH, or multisig nature, excluding `RETURN` and P2SH itself.

As of version 0.9.2 of the Bitcoin Core client, P2SH transactions can contain any valid script, making the P2SH standard much more flexible and allowing for experimentation with many novel and complex types of transactions.

Note that you are not able to put a P2SH inside a P2SH redeem script, because the P2SH specification is not recursive. While it is technically possible to include `RETURN` in a redeem script, as nothing in the rules prevents you from doing so, it is of no

practical use because executing RETURN during validation will cause the transaction to be marked invalid.

Note that because the redeem script is not presented to the network until you attempt to spend a P2SH output, if you lock an output with the hash of an invalid redeem script it will be processed regardless. The UTXO will be successfully locked. However, you will not be able to spend it because the spending transaction, which includes the redeem script, will not be accepted because it is an invalid script. This creates a risk, because you can lock bitcoin in a P2SH that cannot be spent later. The network will accept the P2SH locking script even if it corresponds to an invalid redeem script, because the script hash gives no indication of the script it represents.

P2SH locking scripts contain the hash of a redeem script, which gives no clues as to the content of the redeem script itself. The P2SH transaction will be considered valid and accepted even if the redeem script is invalid. You might accidentally lock bitcoin in such a way that it cannot later be spent.

Data Recording Output (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 nonpayment 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 freeform 20-byte field. Because 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. These transactions that can never be spent are therefore never removed from the UTXO set and cause the size of the UTXO database to forever increase, or "bloat."

In version 0.9 of the Bitcoin Core client, a compromise was reached with the introduction of the RETURN operator. RETURN allows developers to add 80 bytes of nonpay-

ment data to a transaction output. However, unlike the use of "fake" UTXO, the RETURN operator creates an explicitly *provably unspendable* output, which does not need to be stored in the UTXO set. 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.

RETURN scripts look like this:

```
RETURN <data>
```

The data portion is limited to 80 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 Proof of Existence digital notarization service uses the 8-byte prefix DOCPROOF, which is ASCII encoded as `44 4f 43 50 52 4f 4f 46` in hexadecimal.

Keep in mind that there is no "unlocking script" that corresponds to RETURN that could possibly be used to "spend" a RETURN output. The whole point of 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—RETURN is *provably unspendable*. RETURN is usually an output with a zero bitcoin amount, because any bitcoin assigned to such an output is effectively lost forever. If a RETURN is referenced as an input in a transaction, the script validation engine will halt the execution of the validation script and mark the transaction as invalid. The execution of RETURN essentially causes the script to "RETURN" with a FALSE and halt. Thus, if you accidentally reference a RETURN output as an input in a transaction, that transaction is invalid.

A standard transaction (one that conforms to the `isStandard()` checks) can have only one RETURN output. However, a single RETURN output can be combined in a transaction with outputs of any other type.

Two new command-line options have been added in Bitcoin Core as of version 0.10. The option `datacarrier` controls relay and mining of RETURN transactions, with the default set to "1" to allow them. The option `datacarriersize` takes a numeric argument specifying the maximum size in bytes of the RETURN script, 83 bytes by default, which, allows for a maximum of 80 bytes of RETURN data plus one byte of RETURN opcode and two bytes of PUSHDATA opcode.

RETURN was initially proposed with a limit of 80 bytes, but the limit was reduced to 40 bytes when the feature was released. In February 2015, in version 0.10 of Bitcoin Core, the limit was raised back to 80 bytes. Nodes may choose not to relay or mine RETURN, or only relay and mine RETURN containing less than 80 bytes of data.

Timelocks

Timelocks are restrictions on transactions or outputs that only allow spending after a point in time. Bitcoin has had a transaction-level timelock feature from the beginning. It is implemented by the `nLocktime` field in a transaction. Two new timelock features were introduced in late 2015 and mid-2016 that offer UTXO-level timelocks. These are `CHECKLOCKTIMEVERIFY` and `CHECKSEQUENCEVERIFY`.

Timelocks are useful for postdating transactions and locking funds to a date in the future. More importantly, timelocks extend bitcoin scripting into the dimension of time, opening the door for complex multistep smart contracts.

Transaction Locktime (nLocktime)

From the beginning, bitcoin has had a transaction-level timelock feature. Transaction locktime is a transaction-level setting (a field in the transaction data structure) that defines the earliest time that a transaction is valid and can be relayed on the network or added to the blockchain. Locktime is also known as `nLocktime` from the variable name used in the Bitcoin Core codebase. It is set to zero in most transactions to indicate immediate propagation and execution. If `nLocktime` is nonzero and below 500 million, it is interpreted as a block height, meaning the transaction is not valid and is not relayed or 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 valid prior to the specified time. Transactions with `nLocktime` specifying a future block or time must be held by the originating system and transmitted to the bitcoin network only after they become valid. If a transaction is transmitted to the network before the specified `nLocktime`, the transaction will be rejected by the first node as invalid and will not be relayed to other nodes. The use of `nLocktime` is equivalent to postdating a paper check.

Transaction locktime limitations

`nLocktime` has the limitation that while it makes it possible to spend some outputs in the future, it does not make it impossible to spend them until that time. Let's explain that with the following example.

Alice signs a transaction spending one of her outputs to Bob's address, and sets the transaction `nLocktime` to 3 months in the future. Alice sends that transaction to Bob to hold. With this transaction Alice and Bob know that:

- Bob cannot transmit the transaction to redeem the funds until 3 months have elapsed.
- Bob may transmit the transaction after 3 months.

However:

- Alice can create another transaction, double-spending the same inputs without a locktime. Thus, Alice can spend the same UTXO before the 3 months have elapsed.
- Bob has no guarantee that Alice won't do that.

It is important to understand the limitations of transaction `nLocktime`. The only guarantee is that Bob will not be able to redeem it before 3 months have elapsed. There is no guarantee that Bob will get the funds. To achieve such a guarantee, the timelock restriction must be placed on the UTXO itself and be part of the locking script, rather than on the transaction. This is achieved by the next form of timelock, called Check Lock Time Verify.

Check Lock Time Verify (CLTV)

In December 2015, a new form of timelock was introduced to bitcoin as a soft fork upgrade. Based on a specifications in BIP-65, a new script operator called *CHECKLOCKTIMEVERIFY* (*CLTV*) was added to the scripting language. `CLTV` is a per-output timelock, rather than a per-transaction timelock as is the case with `nLocktime`. This allows for much greater flexibility in the way timelocks are applied.

In simple terms, by adding the `CLTV` opcode in the redeem script of an output it restricts the output, so that it can only be spent after the specified time has elapsed.

While `nLocktime` is a transaction-level timelock, `CLTV` is an output-based timelock.

`CLTV` doesn't replace `nLocktime`, but rather restricts specific UTXO such that they can only be spent in a future transaction with `nLocktime` set to a greater or equal value.

The `CLTV` opcode takes one parameter as input, expressed as a number in the same format as `nLocktime` (either a block height or Unix epoch time). As indicated by the `VERIFY` suffix, `CLTV` is the type of opcode that halts execution of the script if the outcome is `FALSE`. If it results in TRUE, execution continues.

In order to lock an output with `CLTV`, you insert it into the redeem script of the output in the transaction that creates the output. For example, if Alice is paying Bob's address, the output would normally contain a P2PKH script like this:

```
DUP HASH160 <Bob's Public Key Hash> EQUALVERIFY CHECKSIG
```

To lock it to a time, say 3 months from now, the transaction would be a P2SH transaction with a redeem script like this:

```
<now + 3 months> CHECKLOCKTIMEVERIFY DROP DUP HASH160 <Bob's Public Key Hash>
EQUALVERIFY CHECKSIG
```

where `<now + 3 months>` is a block height or time value estimated 3 months from the time the transaction is mined: current block height + 12,960 (blocks) or current Unix epoch time + 7,760,000 (seconds). For now, don't worry about the `DROP` opcode that follows `CHECKLOCKTIMEVERIFY`; it will be explained shortly.

When Bob tries to spend this UTXO, he constructs a transaction that references the UTXO as an input. He uses his signature and public key in the unlocking script of that input and sets the transaction `nLocktime` to be equal or greater to the timelock in the `CHECKLOCKTIMEVERIFY` Alice set. Bob then broadcasts the transaction on the bitcoin network.

Bob's transaction is evaluated as follows. If the `CHECKLOCKTIMEVERIFY` parameter Alice set is less than or equal the spending transaction's `nLocktime`, script execution continues (acts as if a "no operation" or NOP opcode was executed). Otherwise, script execution halts and the transaction is deemed invalid.

More precisely, `CHECKLOCKTIMEVERIFY` fails and halts execution, marking the transaction invalid if (source: BIP-65):

1. the stack is empty; or
2. the top item on the stack is less than 0; or
3. the lock-time type (height versus timestamp) of the top stack item and the `nLocktime` field are not the same; or
4. the top stack item is greater than the transaction's `nLocktime` field; or
5. the `nSequence` field of the input is 0xffffffff.

`CLTV` and `nLocktime` use the same format to describe timelocks, either a block height or the time elapsed in seconds since Unix epoch. Critically, when used together, the format of `nLocktime` must match that of `CLTV` in the inputs—they must both reference either block height or time in seconds.

After execution, if `CLTV` is satisfied, the time parameter that preceded it remains as the top item on the stack and may need to be dropped, with `DROP`, for correct execution of subsequent script opcodes. You will often see `CHECKLOCKTIMEVERIFY` followed by `DROP` in scripts for this reason.

By using nLocktime in conjunction with CLTV, the scenario described in "Transaction locktime limitations" on page 157 changes. Because Alice locked the UTXO itself, it is now impossible for either Bob or Alice to spend it before the 3-month locktime has expired.

By introducing timelock functionality directly into the scripting language, CLTV allows us to develop some very interesting complex scripts.

The standard is defined in BIP-65 (CHECKLOCKTIMEVERIFY).

Relative Timelocks

nLocktime and CLTV are both *absolute timelocks* in that they specify an absolute point in time. The next two timelock features we will examine are *relative timelocks* in that they specify, as a condition of spending an output, an elapsed time from the confirmation of the output in the blockchain.

Relative timelocks are useful because they allow a chain of two or more interdependent transactions to be held off chain, while imposing a time constraint on one transaction that is dependent on the elapsed time from the confirmation of a previous transaction. In other words, the clock doesn't start counting until the UTXO is recorded on the blockchain. This functionality is especially useful in bidirectional state channels and Lightning Networks, as we will see in "Payment Channels and State Channels" on page 284.

Relative timelocks, like absolute timelocks, are implemented with both a transaction-level feature and a script-level opcode. The transaction-level relative timelock is implemented as a consensus rule on the value of nSequence, a transaction field that is set in every transaction input. Script-level relative timelocks are implemented with the CHECKSEQUENCEVERIFY (CSV) opcode.

Relative timelocks are implemented according to the specifications in BIP-68, Relative lock-time using consensus-enforced sequence numbers and BIP-112, CHECKSEQUENCEVERIFY.

BIP-68 and BIP-112 were activated in May 2016 as a soft fork upgrade to the consensus rules.

Relative Timelocks with nSequence

Relative timelocks can be set on each input of a transaction, by setting the nSequence field in each input.

Original meaning of nSequence

The nSequence field was originally intended (but never properly implemented) to allow modification of transactions in the mempool. In that use, a transaction contain-

ing inputs with nSequence value below 2^{32} (0xFFFFFFFF) indicated a transaction that was not yet "finalized." Such a transaction would be held in the mempool until it was replaced by another transaction spending the same inputs with a higher nSequence value. Once a transaction was received whose inputs had an nSequence value of 2^{32} it would be considered "finalized" and mined.

The original meaning of nSequence was never properly implemented and the value of nSequence is customarily set to 2^{32} in transactions that do not utilize timelocks. For transactions with nLocktime or CHECKLOCKTIMEVERIFY, the nSequence value must be set to less than 2^{32} for the timelock guards to have effect. Customarily, it is set to $2^{32} - 1$ (0xFFFFFFFE).

nSequence as a consensus-enforced relative timelock

Since the activation of BIP-68, new consensus rules apply for any transaction containing an input whose nSequence value is less than 2^{31} (bit 1<<31 is not set). Programmatically, that means that if the most significant (bit 1<<31) is not set, it is a flag that means "relative locktime." Otherwise (bit 1<<31 set), the nSequence value is reserved for other uses such as enabling CHECKLOCKTIMEVERIFY, nLocktime, Opt-In-Replace-By-Fee, and other future developments.

Transaction inputs with nSequence values less than 2^{31} are interpreted as having a relative timelock. Such a transaction is only valid once the input has aged by the relative timelock amount. For example, a transaction with one input with an nSequence relative timelock of 30 blocks is only valid when at least 30 blocks have elapsed from the time the UTXO referenced in the input was mined. Since nSequence is a per-input field, a transaction may contain any number of timelocked inputs, all of which must have sufficiently aged for the transaction to be valid. A transaction can include both timelocked inputs (nSequence < 2^{31}) and inputs without a relative timelock (nSequence >= 2^{31}).

The nSequence value is specified in either blocks or seconds, but in a slightly different format than we saw used in nLocktime. A type-flag is used to differentiate between values counting blocks and values counting time in seconds. The type-flag is set in the 23rd least-significant bit (i.e., value 1<<22). If the type-flag is set, then the nSequence value is interpreted as a multiple of 512 seconds. If the type-flag is not set, the nSequence value is interpreted as a number of blocks.

When interpreting nSequence as a relative timelock, only the 16 least significant bits are considered. Once the flags (bits 32 and 23) are evaluated, the nSequence value is usually "masked" with a 16-bit mask (e.g., nSequence & 0x0000FFFF).

Figure 7-1 shows the binary layout of the nSequence value, as defined by BIP-68.

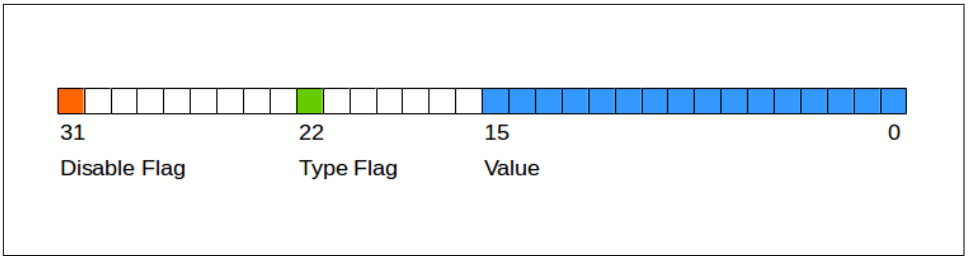

Figure 7-1. BIP-68 definition of nSequence encoding (Source: BIP-68)

Relative timelocks based on consensus enforcement of the `nSequence` value are defined in BIP-68.

The standard is defined in BIP-68, Relative lock-time using consensus-enforced sequence numbers.

Relative Timelocks with CSV

Just like CLTV and `nLocktime`, there is a script opcode for relative timelocks that leverages the `nSequence` value in scripts. That opcode is `CHECKSEQUENCEVERIFY`, commonly referred to as `CSV` for short.

The `CSV` opcode when evaluated in a UTXO's redeem script allows spending only in a transaction whose input `nSequence` value is greater than or equal to the `CSV` parameter. Essentially, this restricts spending the UTXO until a certain number of blocks or seconds have elapsed relative to the time the UTXO was mined.

As with CLTV, the value in `CSV` must match the format in the corresponding `nSequence` value. If `CSV` is specified in terms of blocks, then so must `nSequence`. If `CSV` is specified in terms of seconds, then so must `nSequence`.

Relative timelocks with `CSV` are especially useful when several (chained) transactions are created and signed, but not propagated, when they're kept "off-chain." A child transaction cannot be used until the parent transaction has been propagated, mined, and aged by the time specified in the relative timelock. One application of this use case can be seen in "Payment Channels and State Channels" on page 284 and "Routed Payment Channels (Lightning Network)" on page 297.

`CSV` is defined in detail in BIP-112, CHECKSEQUENCEVERIFY.

Median-Time-Past

As part of the activation of relative timelocks, there was also a change in the way "time" is calculated for timelocks (both absolute and relative). In bitcoin there is a subtle, but very significant, difference between wall time and consensus time. Bitcoin

is a decentralized network, which means that each participant has his or her own perspective of time. Events on the network do not occur instantaneously everywhere. Network latency must be factored into the perspective of each node. Eventually everything is synchronized to create a common ledger. Bitcoin reaches consensus every 10 minutes about the state of the ledger as it existed in the *past*.

The timestamps set in block headers are set by the miners. There is a certain degree of latitude allowed by the consensus rules to account for differences in clock accuracy between decentralized nodes. However, this creates an unfortunate incentive for miners to lie about the time in a block so as to earn extra fees by including timelocked transactions that are not yet mature. See the following section for more information.

To remove the incentive to lie and strengthen the security of timelocks, a BIP was proposed and activated at the same time as the BIPs for relative timelocks. This is BIP-113, which defines a new consensus measurement of time called *Median-Time-Past*.

Median-Time-Past is calculated by taking the timestamps of the last 11 blocks and finding the median. That median time then becomes consensus time and is used for all timelock calculations. By taking the midpoint from approximately two hours in the past, the influence of any one block's timestamp is reduced. By incorporating 11 blocks, no single miner can influence the timestamps in order to gain fees from transactions with a timelock that hasn't yet matured.

Median-Time-Past changes the implementation of time calculations for `nLocktime`, `CLTV`, `nSequence`, and `CSV`. The consensus time calculated by Median-Time-Past is always approximately one hour behind wall clock time. If you create timelock transactions, you should account for it when estimating the desired value to encode in `nLocktime`, `nSequence`, `CLTV`, and `CSV`.

Median-Time-Past is specified in BIP-113.

Timelock Defense Against Fee Sniping

Fee-sniping is a theoretical attack scenario, where miners attempting to rewrite past blocks "snipe" higher-fee transactions from future blocks to maximize their profitability.

For example, let's say the highest block in existence is block #100,000. If instead of attempting to mine block #100,001 to extend the chain, some miners attempt to remine #100,000. These miners can choose to include any valid transaction (that hasn't been mined yet) in their candidate block #100,000. They don't have to remine the block with the same transactions. In fact, they have the incentive to select the most profitable (highest fee per kB) transactions to include in their block. They can include any transactions that were in the "old" block #100,000, as well as any transac-

tions from the current mempool. Essentially they have the option to pull transactions from the "present" into the rewritten "past" when they re-create block #100,000.

Today, this attack is not very lucrative, because block reward is much higher than total fees per block. But at some point in the future, transaction fees will be the majority of the reward (or even the entirety of the reward). At that time, this scenario becomes inevitable.

To prevent "fee sniping," when Bitcoin Core creates transactions, it uses `nLocktime` to limit them to the "next block," by default. In our scenario, Bitcoin Core would set `nLocktime` to 100,001 on any transaction it created. Under normal circumstances, this `nLocktime` has no effect—the transactions could only be included in block #100,001 anyway; it's the next block.

But under a blockchain fork attack, the miners would not be able to pull high-fee transactions from the mempool, because all those transactions would be timelocked to block #100,001. They can only remine #100,000 with whatever transactions were valid at that time, essentially gaining no new fees.

To achieve this, Bitcoin Core sets the `nLocktime` on all new transactions to <current block # + 1> and sets the `nSequence` on all the inputs to 0xFFFFFFFE to enable `nLocktime`.

Scripts with Flow Control (Conditional Clauses)

One of the more powerful features of Bitcoin Script is flow control, also known as conditional clauses. You are probably familiar with flow control in various programming languages that use the construct `IF...THEN...ELSE`. Bitcoin conditional clauses look a bit different, but are essentially the same construct.

At a basic level, bitcoin conditional opcodes allow us to construct a redeem script that has two ways of being unlocked, depending on a `TRUE`/`FALSE` outcome of evaluating a logical condition. For example, if x is `TRUE`, the redeem script is A and the ELSE redeem script is B.

Additionally, bitcoin conditional expressions can be "nested" indefinitely, meaning that a conditional clause can contain another within it, which contains another, etc. Bitcoin Script flow control can be used to construct very complex scripts with hundreds or even thousands of possible execution paths. There is no limit to nesting, but consensus rules impose a limit on the maximum size, in bytes, of a script.

Bitcoin implements flow control using the `IF`, `ELSE`, `ENDIF`, and `NOTIF` opcodes. Additionally, conditional expressions can contain boolean operators such as `BOOLAND`, `BOOLOR`, and `NOT`.

At first glance, you may find the bitcoin's flow control scripts confusing. That is because Bitcoin Script is a stack language. The same way that `1 + 1` looks "backward" when expressed as `1 1 ADD`, flow control clauses in bitcoin also look "backward."

In most traditional (procedural) programming languages, flow control looks like this:

```
if (condition):
  code to run when condition is true
else:
  code to run when condition is false
code to run in either case
```

In a stack-based language like Bitcoin Script, the logical condition comes before the `IF`, which makes it look "backward," like this:

```
condition
IF
  code to run when condition is true
ELSE
  code to run when condition is false
ENDIF
code to run in either case
```

When reading Bitcoin Script, remember that the condition being evaluated comes *before* the `IF` opcode.

Conditional Clauses with VERIFY Opcodes

Another form of conditional in Bitcoin Script is any opcode that ends in `VERIFY`. The `VERIFY` suffix means that if the condition evaluated is not `TRUE`, execution of the script terminates immediately and the transaction is deemed invalid.

Unlike an `IF` clause, which offers alternative execution paths, the `VERIFY` suffix acts as a *guard clause*, continuing only if a precondition is met.

For example, the following script requires Bob's signature and a pre-image (secret) that produces a specific hash. Both conditions must be satisfied to unlock:

```
HASH160 <expected hash> EQUALVERIFY <Bob's Pubkey> CHECKSIG
```

To redeem this, Bob must construct an unlocking script that presents a valid pre-image and a signature:

```
<Bob's Sig> <hash pre-image>
```

Without presenting the pre-image, Bob can't get to the part of the script that checks for his signature.

This script can be written with an `IF` instead:

```
HASH160 <expected hash> EQUAL
IF
```

```
    <Bob's Pubkey> CHECKSIG
ENDIF
```

Bob's unlocking script is identical:

```
<Bob's Sig> <hash pre-image>
```

The script with `IF` does the same thing as using an opcode with a `VERIFY` suffix; they both operate as guard clauses. However, the `VERIFY` construction is more efficient, using one fewer opcode.

So, when do we use `VERIFY` and when do we use `IF`? If all we are trying to do is to attach a precondition (guard clause), then `VERIFY` is better. If, however, we want to have more than one execution path (flow control), then we need an `IF…ELSE` flow control clause.

An opcode such as `EQUAL` will push the result (`TRUE`/`FALSE`) onto the stack, leaving it there for evaluation by subsequent opcodes. In contrast, the opcode `EQUALVERIFY` suffix does not leave anything on the stack. Opcodes that end in `VERIFY` do not leave the result on the stack.

Using Flow Control in Scripts

A very common use for flow control in Bitcoin Script is to construct a redeem script that offers multiple execution paths, each a different way of redeeming the UTXO.

Let's look at a simple example, where we have two signers, Alice and Bob, and either one is able to redeem. With multisig, this would be expressed as a 1-of-2 multisig script. For the sake of demonstration, we will do the same thing with an `IF` clause:

```
IF
 <Alice's Pubkey> CHECKSIG
ELSE
 <Bob's Pubkey> CHECKSIG
ENDIF
```

Looking at this redeem script, you may be wondering: "Where is the condition? There is nothing preceding the `IF` clause!"

The condition is not part of the redeem script. Instead, the condition will be offered in the unlocking script, allowing Alice and Bob to "choose" which execution path they want.

Alice redeems this with the unlocking script:

```
<Alice's Sig> 1
```

The `1` at the end serves as the condition (`TRUE`) that will make the `IF` clause execute the first redemption path for which Alice has a signature.

For Bob to redeem this, he would have to choose the second execution path by giving a FALSE value to the IF clause:

```
<Bob's Sig> 0
```

Bob's unlocking script puts a 0 on the stack, causing the IF clause to execute the second (ELSE) script, which requires Bob's signature.

Since IF clauses can be nested, we can create a "maze" of execution paths. The unlocking script can provide a "map" selecting which execution path is actually executed:

```
IF
  script A
ELSE
  IF
    script B
  ELSE
    script C
  ENDIF
ENDIF
```

In this scenario, there are three execution paths (script A, script B, and script C). The unlocking script provides a path in the form of a sequence of TRUE or FALSE values. To select path script B, for example, the unlocking script must end in 1 0 (TRUE, FALSE). These values will be pushed onto the stack, so that the second value (FALSE) ends up at the top of the stack. The outer IF clause pops the FALSE value and executes the first ELSE clause. Then the TRUE value moves to the top of the stack and is evaluated by the inner (nested) IF, selecting the B execution path.

Using this construct, we can build redeem scripts with tens or hundreds of execution paths, each offering a different way to redeem the UTXO. To spend, we construct an unlocking script that navigates the execution path by putting the appropriate TRUE and FALSE values on the stack at each flow control point.

Complex Script Example

In this section we combine many of the concepts from this chapter into a single example.

Our example uses the story of Mohammed, the company owner in Dubai who is operating an import/export business.

In this example, Mohammed wishes to construct a company capital account with flexible rules. The scheme he creates requires different levels of authorization depending on timelocks. The participants in the multisig scheme are Mohammed, his two partners Saeed and Zaira, and their company lawyer Abdul. The three partners make decisions based on a majority rule, so two of the three must agree. However, in the

case of a problem with their keys, they want their lawyer to be able to recover the funds with one of the three partner signatures. Finally, if all partners are unavailable or incapacitated for a while, they want the lawyer to be able to manage the account directly.

Here's the script that Mohammed designs to achieve this:

```
IF
  IF
    2
  ELSE
    <30 days> CHECKSEQUENCEVERIFY DROP
    <Abdul the Lawyer's Pubkey> CHECKSIGVERIFY
    1
  ENDIF
  <Mohammed's Pubkey> <Saeed's Pubkey> <Zaira's Pubkey> 3 CHECKMULTISIG
ELSE
  <90 days> CHECKSEQUENCEVERIFY DROP
  <Abdul the Lawyer's Pubkey> CHECKSIG
ENDIF
```

Mohammed's script implements three execution paths using nested `IF…ELSE` flow control clauses.

In the first execution path, this script operates as a simple 2-of-3 multisig with the three partners. This execution path consists of lines 3 and 9. Line 3 sets the quorum of the multisig to `2` (2-of-3). This execution path can be selected by putting `TRUE TRUE` at the end of the unlocking script:

```
0 <Mohammed's Sig> <Zaira's Sig> TRUE TRUE
```

The `0` at the beginning of this unlocking script is because of a bug in `CHECKMULTISIG` that pops an extra value from the stack. The extra value is disregarded by the `CHECKMULTISIG`, but it must be present or the script fails. Pushing `0` (customarily) is a workaround to the bug, as described in "A bug in CHECKMULTISIG execution" on page 150.

The second execution path can only be used after 30 days have elapsed from the creation of the UTXO. At that time, it requires the signature of Abdul the lawyer and one of the three partners (a 1-of-3 multisig). This is achieved by line 7, which sets the quorum for the multisig to `1`. To select this execution path, the unlocking script would end in `FALSE TRUE`:

```
0 <Saeed's Sig> <Abdul's Sig> FALSE TRUE
```

Why `FALSE TRUE`? Isn't that backward? Because the two values are pushed on to the stack, with `FALSE` pushed first, then `TRUE` pushed second. `TRUE` is therefore popped *first* by the first `IF` opcode.

Finally, the third execution path allows Abdul the lawyer to spend the funds alone, but only after 90 days. To select this execution path, the unlocking script has to end in `FALSE`:

```
<Abdul's Sig> FALSE
```

Try running the script on paper to see how it behaves on the stack.

A few more things to consider when reading this example. See if you can find the answers:

- Why can't the lawyer redeem the third execution path at any time by selecting it with `FALSE` on the unlocking script?
- How many execution paths can be used 5, 35, and 105 days, respectively, after the UTXO is mined?
- Are the funds lost if the lawyer loses his key? Does your answer change if 91 days have elapsed?
- How do the partners "reset" the clock every 29 or 89 days to prevent the lawyer from accessing the funds?
- Why do some `CHECKSIG` opcodes in this script have the `VERIFY` suffix while others don't?