Introduction to Bitcoin Script

Chapter 1: About Bitcoin Script

01 - Introduction

Welcome to Development with Script. In this course we will focus on Bitcoin Script, the immensely powerful and flexible programming language that is used in all Bitcoin transactions. Script is a fascinating tool which allows complex transactional conditions to be developed, using the set-in-stone Bitcoin protocol. This course is aimed at giving you a solid grasp of the ways in which Bitcoin script's unique functionality is applied within Bitcoin transactions to allow you to build applications that leverage the superpowers of Bitcoin.

Insert Chapter 1 video 1

The course will be broken down into 5 chapters as follows:

Chapter 1 will focus on the history of FORTH and its relationship to Bitcoin Script, and the differences between the two. It will also look at the efficiency of script which is a key element of the Bitcoin protocol that will eventually allow nodes to validate hundreds of millions of transactions per second. We will also discuss why script does not allow jumps in code, but also look at how it can be used to implement turing complete systems. We will also have a look at how Bitcoin Script has evolved over the years to get to its current state, and discuss why there will be no further changes to its design.

Chapter 2 will begin looking at the programming language itself, in particular some of the rules that are applied when scripts are validated. We will then look at the capabilities of script, and explore how it might be used as a basis to implement more general purpose computing systems. We will also discuss the architecture of the script evaluator, including the two stacks, how they differ, and the purpose each can be applied to, as well as looking in-depth at reverse polish notation, and discussing how it is employed in Bitcoin, and some of the requirements around optimising data item usage.

Chapter 3 will take students through the Bitcoin Script opcodes. The opcodes can be considered a library of primitive functions that are used to express the desired functionality required of a script. Bitcoin's opcodes can be used to build any compute function possible with a little imagination. We will also take a brief look back at the changes that have been rolled into Bitcoin Script, and what they might mean for anyone who has written scripts to the ledger using older variations of the opcode set.

Chapter 4 will take a closer look at a specific set of opcodes that offer higher level functionality, including IF/ELSE conditions, string formatting, and most importantly, the digital signature checks that keep Bitcoin outputs secure. We will take a close look into what a digital signature checks and how.

Chapter 5 will then look at a series of well-known and battle-tested transaction types, to give you an idea of how templates can be employed to deliver applications to millions of unique users.

We hope that this course will give you the knowledge and skills you need to be more effective in your development of applications on Bitcoin, and allows you to build better products that serve more people faster.

Thank you.

Chapter 1 video 1

Script	Video
Welcome to Introduction to Bitcoin script. In this course, we will take an in-depth look at the programming language that gives Bitcoin transactions their unique and flexible functionality.	Evan's face
Bitcoin Script is the versatile programming language at the core of the Bitcoin protocol.	Cool animation of the protocol
This course will take you through the history of Bitcoin Script	Evan
how it is expressed and then look in detail at the complete opcode set.	scroll through OPCODE list
Once these foundations are set, we will look at a series of simple script examples	Evan
finishing with a detailed overview of OP_PUSH_TX and how it can be used to control the outcome of a transaction.	Golden 3D 'OP_PUSH_TX' with arrows coming pointing to 'Internet of Things' 'Control Systems' 'Government Services' 'Gaming' 'Apps and products' 'Distributed Computing Services' 'Machine Learning'
These core skills should give you everything you need to begin your journey as a Bitcoin Script engineer.	Evan

02 - FORTH: A Precursor to Bitcoin Script

In the late 60's Charles 'Chuck' Moore developed FORTH as a fully interactive stack based programming environment that ran on a microcontroller and provided the user with a simple, command line entry based means to enter commands and build FORTH words and programs. FORTH is different to many programming languages in that it runs 'live' while the programmer is working on their software, allowing new 'words' to be defined, tested, redefined and debugged without having to recompile or restart the system. All elements of the syntax are defined as words, including variables such as constants and basic operators and the system uses a simple stack to pass instructions between words.

The following video is a 2 minute overview of Forth.

Embed: https://www.youtube.com/watch?v=ml9s2HfpDZY

Stack based programming such as that which is used in FORTH is called Reverse Polish Notation (RPN). In RPN, the operands needed for a function must be pushed onto the stack *prior* to the word that processes them being called. Data types are fluid, with each item existing as a bytevector in memory. Each function uses the data items it expects to see. A function will treat the bytevector however it is programmed to. It may be seen as an integer, a string of text, a floating point number or any other type of data. If a word performs a function that consumes more than one data point, it will expect the right number of items of the right types to be on the stack.

For example, to multiply 2 by 3, the user would input '2 3 *', or '2 3 multiply'. This would consume the 2 and 3 from the stack and replace them with a 6. If there are less than 2 items, or they are too big to be integers, the program ends with an error.

All forth words are built from a set of primitive words. These are built upon to create application specific functionality that can easily interact with microprocessor input/output hardware. Digital and analog signals can be evaluated, and techniques such as fast fourier transforms can be implemented to filter and read real world data. This ensures a high degree of efficiency and flexibility for any given system or architecture. A basic set of words can be extended to create complex applications in a tiny space.

03 - From FORTH to Bitcoin Script

FORTH is the parent of Bitcoin Script, which adds some grammar changes and removes capabilities such as jump instructions and loops.

In transactions on the Bitcoin ledger, each output contains a predicate called a scriptPubKey. This predicate is written in Bitcoin script and evaluates an input against a set of conditions. When a transaction output is spent, the user provides a set of input data called the scriptSig which is loaded onto the stack before the scriptPubKey processes it. If the evaluation finishes with a single non-zero value on the stack, the scriptSig is valid and the tokens contained in the output can be spent in the transaction. For a transaction to be valid, every input must have a valid scriptSig and every new scriptPubKey must meet the Bitcoin grammar rules we will cover in chapter 2.

Insert chapter 1 video 2

Bitcoin script predicates can only contain opcodes defined within the Bitcoin protocol itself. The evaluation ends when any of the following conditions are met:

- The script reaches an OP_RETURN opcode
- The script fails an OP_VERIFY check
- The scriptSig is incomplete or invalid
- The end of the script is reached
- The script exceeds a policy such as the maximum stack memory policy (for more information on policies, please see the BitcoinSV wiki, or consider taking Introduction to Bitcoin Infrastructure)

Bitcoin script cannot jump back to a previous instruction. For an input to a transaction to be valid its script must process without halting until it ends.

Chapter 1 video 2

A video on the basics of Bitcoin transactions

Script	Video
Bitcoin transactions can be described as spending coins that were created in previous transactions as inputs, and using the satoshis they contain to create new coins.	Evan's head
Each coin consists of a value in satoshis and a lockScript.	Visualisation of a transaction with 3 outputs, eac containing a value and a lockScript
When a user creates a new transaction, they reference the specific output from a previous transaction that they intend to spend	Show a new transaction being created where it imports one of its inputs from one of the outputs an earlier transaction.
and supply a valid solution to its lockScript. This solution is called the unlockScript (or scriptSig) and it typically includes the spending party's digital signature.	Show that the unlockScript is included in the inp with a signature.
In the Bitcoin evaluator, the unlockScript is loaded onto the stack and then processed using the Bitcoin script contained in the lockScript.	Detatch unlockScript and lockScript, swap from right to left and join. Show evaluation bar.
For a transaction to be valid, all of its inputs must have valid unlockScripts.	Show all inputs to new transaction solving true

04 - Bitcoin's Transaction Protocol

Part of the Bitcoin protocol is a rigid definition of the makeup of a Bitcoin transaction.

The conditions which determine whether transactions are valid or not are governed by rules that we will look at in more detail in subsequent chapters. For now, we will look at the structure of the transactions themselves with a view to understanding how the Bitcoin script itself is evaluated in the process of spending coins on the network.

All transactions are defined using a serialisation format comprised of a number of fixed and variable length fields concatenated into a single string. This can be likened to a Protobuffer.

Transaction are created using the following elements:

1. Version No. (4 bytes)
2. Quantity of transaction inputs (varInt, 1-9 bytes) ← link to Varint page on BSV wiki
3. The list of inputs in input structure format which is defined as:
    1. The TXID the input is from (32 byte little endian hash)
    2. The output index of the input (4 byte little endian integer)
    3. The length of the scriptSig (varInt, 1-9 bytes)
    4. The scriptSig
    5. The input's nSequence value (4 byte little endian integer)
4. Quantity of transaction outputs (varInt, 1-9 bytes)
5. The list of outputs in output structure format which is defined as:
    1. The output's satoshi value (8 byte little endian integer)
    2. The length of the scriptPubKey
    3. The scriptPubKey
6. The transaction's nLocktime

The transaction itself is all of these elements serialised as a bytevector.

To see this, let us look at an example of a Bitcoin transaction.

This is the public record of Satoshi Nakamoto sending Hal Finney 10 Bitcoins in block no. 170. You can find the transaction in the whatssonchain block explorer here. When represented as a hexadecimal string, the transaction looks like this:

```
0100000001c997a5e56e104102fa209c6a852dd90660a20b2d9c352423edce25857fcd3704
000000004847304402204e45e16932b8af514961a1d3a1a25fdf3f4f7732e9d624c6c61548a
b5fb8cd410220181522ec8eca07de4860a4acdd12909d831cc56cbbac4622082221a8768d1d
0901ffffffff0200ca9a3b00000000434104ae1a62fe09c5f51b13905f07f06b99a2f7159b2
225f374cd378d71302fa28414e7aab37397f554a7df5f142c21c1b7303b8a0626f1baded5c7
2a704f7e6cd84cac00286bee0000000043410411db93e1dcdb8a016b49840f8c53bc1eb68a3
82e97b1482ecad7b148a6909a5cb2e0eaddfb84ccf9744464f82e160bfa9b8b64f9d4c03f99
9b8643f656b412a3ac00000000
```

Chapter 1 assessment 1

Go to https://bitcoinsv.academy/hash-calculator

A Bitcoin transaction ID is the double SHA256 hash of the serialised transaction string. The first hash of the string is called the 'intermediate' hash.

Take the serialised transaction from above, and generate the intermediate hash (you must tell it that the data is hex, rather than text). If you take that output and hash it again, and you will see the result is the TXID of the transaction above in little endian format.

1. Which is the intermediate hash?

- [ ] 6abf71178f3ab0eb9ea0fe98dd496c25f54420c1a6e340b6deb7c2fd53226aea
- [ ] 669b8e3b31c51b2fe65c25943a0f1f64c44fcdb3facd8a808303d878d8b02701
- [ ] 169e1e83e930853391bc6f35f605c6754cfead57cf8387639d3b4096c54f18f4
- [ ] 240cf324ec3cf59609733e2a45e1408673306be8dcd4caf3067aa9355a0269e3

05 - Transaction Breakdown

Now we will break down the serialised transaction into its various components.

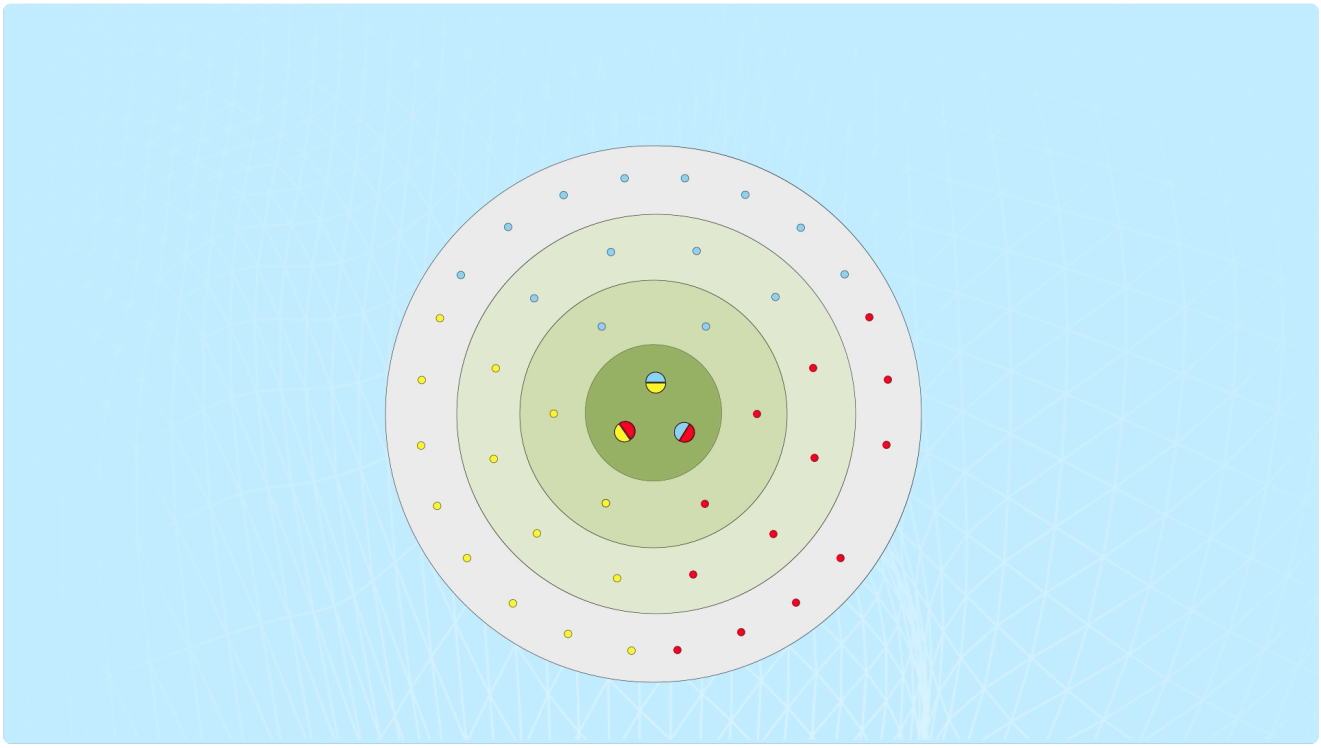
Version

The first 4 bytes are the version number recorded in little endian:

`01000000`

This indicates that the transaction follows the version 01 of the transaction evaluation format.

As usage increases, it is expected this field will be used to signal a variety of different transaction templates which may each be processed by different subsets of the mining network.



Quantity of inputs

`01`

This indicates that there is just 1 input being spent in this transaction.

This value is a 4-byte integer in Little Endian format. This means that if a transaction has more than 0xFFFFFFFF outputs (approx 4.3 billion) those that are created at index locations outside the range are practically unspendable.

The Input

As explained previously, the input itself is broken down into 5 elements as follows:

Previous TXID

`c997a5e56e104102fa209c6a852dd90660a20b2d9c352423edce25857fcd3704`

This is a little endian representation of the input's parent transaction. This indicates that the spending party is using an output of TXID no.
0437cd7f8525ceed2324359c2d0ba26006d92d856a9c20fa0241106ee5a597c9

Output index

`00000000`

This indicates that the spender is using the 'zeroth' output of the transaction.

Extra detail:

The scriptPubKey contained in this output is the following simple Pay to Public Key script:

`0411db93e1dcdb8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5cb2e0eaddfb84ccf9744464f82e160bfa9b8b64f9d4c03f999b8643f656b412a3 OP_CHECKSIG`

The first part of this output is a public key controlled by the output's owner, and the second part is a Bitcoin Script OPCODE that checks the transaction's signature.

Length of the scriptSig

`48`

This value is the length of the scriptSig represented as a hexadecimal value. In this case, 0x48 is the equivalent of 72 in base 10.

scriptSig

`47304402204e45e16932b8af514961a1d3a1a25fdf3f4f7732e9d624c6c61548ab5fb8cd41`
`0220181522ec8eca07de4860a4acdd12909d831cc56cbbac4622082221a8768d1d0901`

The scriptSig in this case is a single pushdata opcode (0x47) which pushes 71 bytes to the stack. These are a single bytevector containing the 70 byte long ECDSA signature in DER format concatenated with the 1 byte SIGHASH flag on the end. In some cases the DER signature is 71 bytes with an additional 1 byte for the SIGHASH flag. To gain a better understanding of the DER signature format and its application to Bitcoin, please consider doing the Bitcoin Primitives course on 'Digital Signatures'.

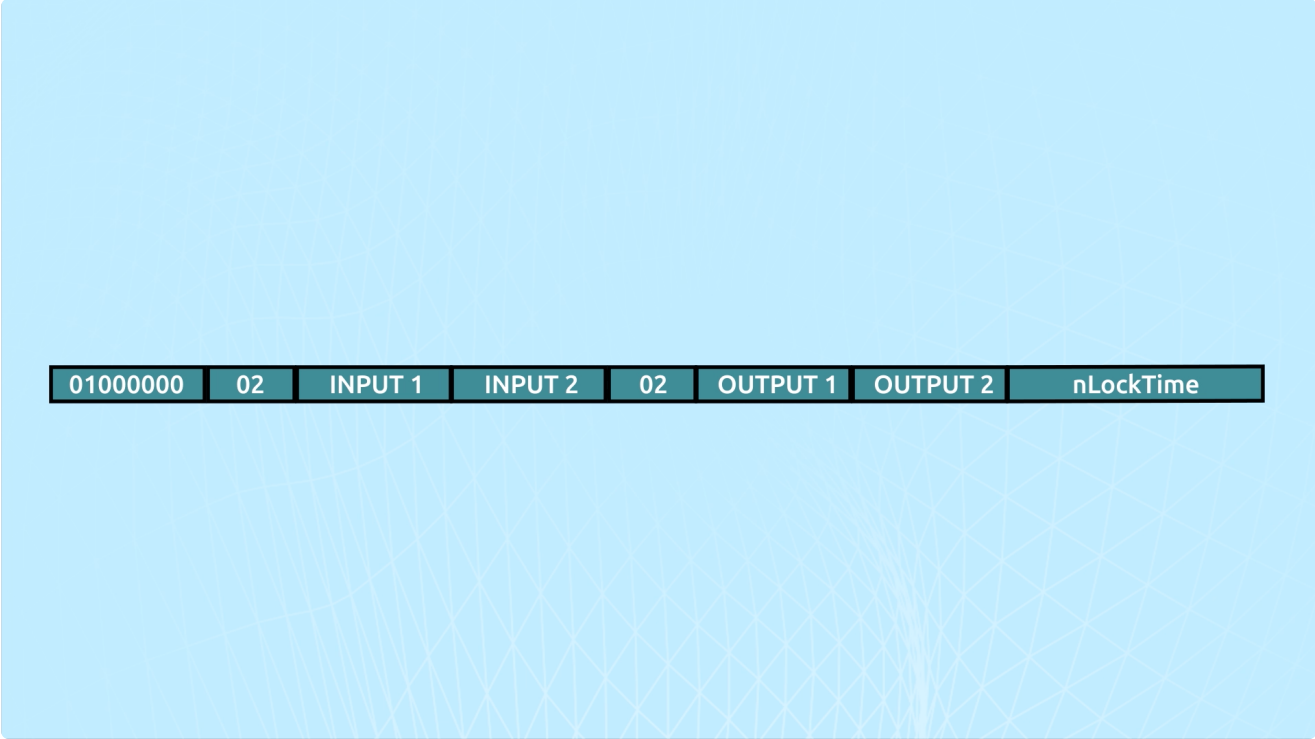
When the evaluation engine validates this transaction, this signature is loaded onto the processing stack first, and then followed with the scriptPubKey contained in the output being spent. In this case, the signature can be shown to have been generated with a private key that corresponds to the public key in the scriptPubKey, meaning the coin could be spent.

nSequence

`ffffffff`

The nSequence value can be used to generate payment channels which allow non-final transactions to be modified many times. In this case, the nSequence number is UINT_MAX which means the transaction was final when it was submitted to the network.

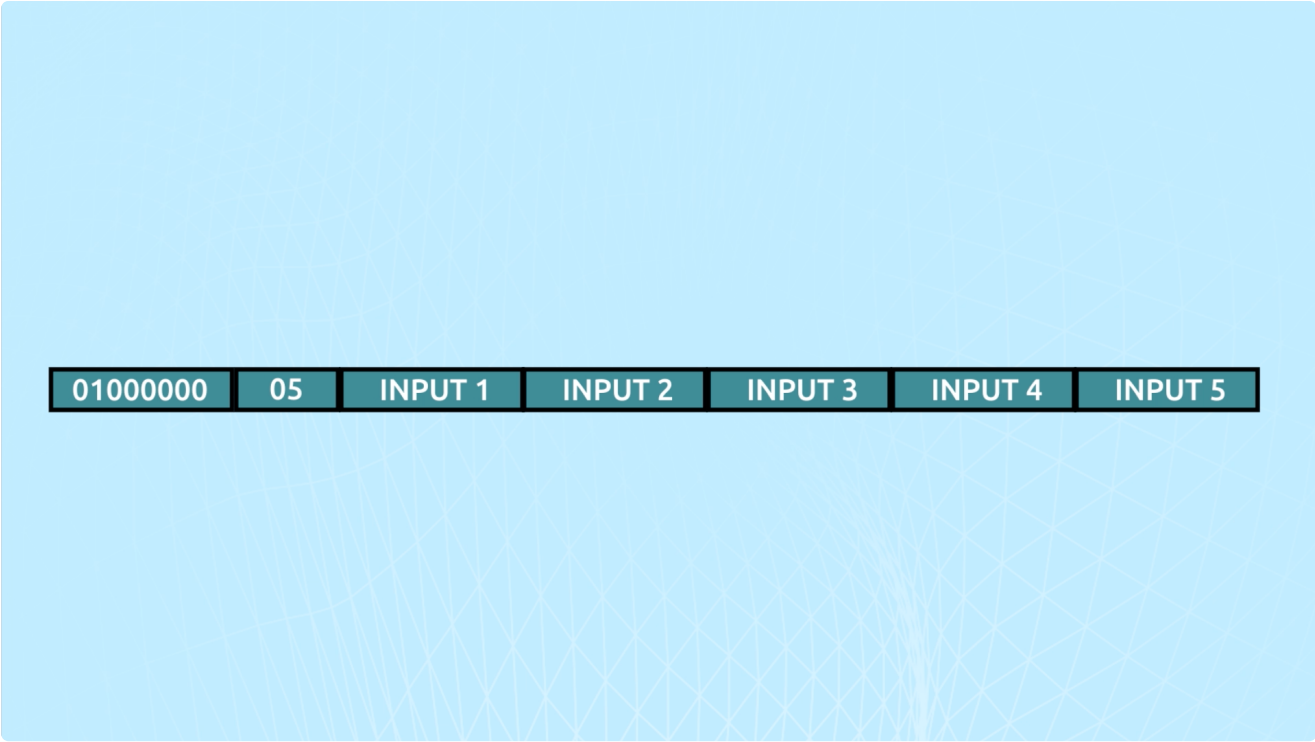
If the nSequence value is not UINT_MAX then the transaction cannot be processed until the nLockTime (expressed as either block height or UNIX epoch) has expired.



Number of Outputs

`02`

This tells us that there are two outputs in this transaction.



This value is a Variable Integer (VarInt) of 1-9 bytes meaning that a transaction can have 1.8 x 10^19 outputs. Remember though, only the first 4.3 billion can be referenced as inputs.

A VarInt is most commonly a 1 byte hexadecimal value:

However, if the VarInt is going to be greater than `0xfc` (so the number you're trying to express won't fit inside of two hexadecimal characters) then you can expand the field in the following way:

No. of inputs	Example	Description
<= `0xfc`	`12`	
`0xfc` < qty <= `0xffff`	**`fd`** `1234`	Prefix with `fd`, and the next *2 bytes* is the VarInt (in little-endian).
`0xffff` < qty <= `0xffffffff`	**`fe`** `12345678`	Prefix with `fe`, and the next *4 bytes* is the VarInt (in little-endian).
`0xffffffff` < qty <= `0xffffffffffffffff`	**`ff`** `1234567890abcdef`	Prefix with `ff`, and the next *8 bytes* is the VarInt (in little-endian).

Output 1 Breakdown

This is the output in which Satoshi sends Hal Finney his Bitcoin.

As explained previously, the first output is broken down into 3 elements as follows:

Output Value

`00ca9a3b00000000`

This value is a little endian integer representing the satoshi value of the first transaction output. In this case it can be expressed in Hexadecimal as 0x000000003b9aca00 which is 1,000,000,000 in decimal. This shows that the first output was sending 10 Bitcoins (each 100,000,000 satoshis) to a new scriptPubKey. In this case, we know that this was a scriptPubKey given to Hal Finney by Satoshi Nakamoto, who sent him the 10 Bitcoins.

Length of the scriptPubKey

`43`

This value is a hexadecimal representation of the length of the scriptPubKey, indicating that it is 0x43, or 67 bytes long.

scriptPubKey

`4104ae1a62fe09c5f51b13905f07f06b99a2f7159b2225f374cd378d71302fa28414e7aab37397f554a7df5f142c21c1b7303b8a0626f1baded5c72a704f7e6cd84cac`

This scriptPubKey represents a predicate expressed in Bitcoin Script. The predicate is broken down as follows:

The first byte (0x41) is a pushdata OPCODE which pushes Hal Finney's 65 byte long public key onto the stack.

The next 65 bytes are Hal Finney's public key as follows:

`04ae1a62fe09c5f51b13905f07f06b99a2f7159b2225f374cd378d71302fa28414e7aab37397f554a7df5f142c21c1b7303b8a0626f1baded5c72a704f7e6cd84c`

The final byte (0xac) is the opcode OP_CHECKSIG.

This represents the most simple application of an ECDSA signature check in Bitcoin.

This scriptPubKey locks the 1,000,000,000 satoshis contained in this output until someone who controls the public key provides the needed signature as an input's scriptSig.

Chapter 1 assessment 2

Using the Output 2 data:

`00286bee00000000434104 11db93e1dcdb8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5cb2e0eaddfb84ccf9744464f82e160bfa9b8b64f9d4c03f999b8643f656b412a3`

Break down the string and provide analysis as outlined in the table:

Output Value	Output value in Satoshis	Length of scriptPubKey (in hexadecimal)	Length of Hal's public key (in Hexadecimal)	Hal's public ke
00000000ee6b2800	4,000,000,000	43	41	0411db93e1d 8a016b49840 53bc1eb68a3 97b1482ecad 48a6909a5cb eaddfb84ccf9 464f82e160bf 8b64f9d4c03f b8643f656b41 3

06 - nLockTime

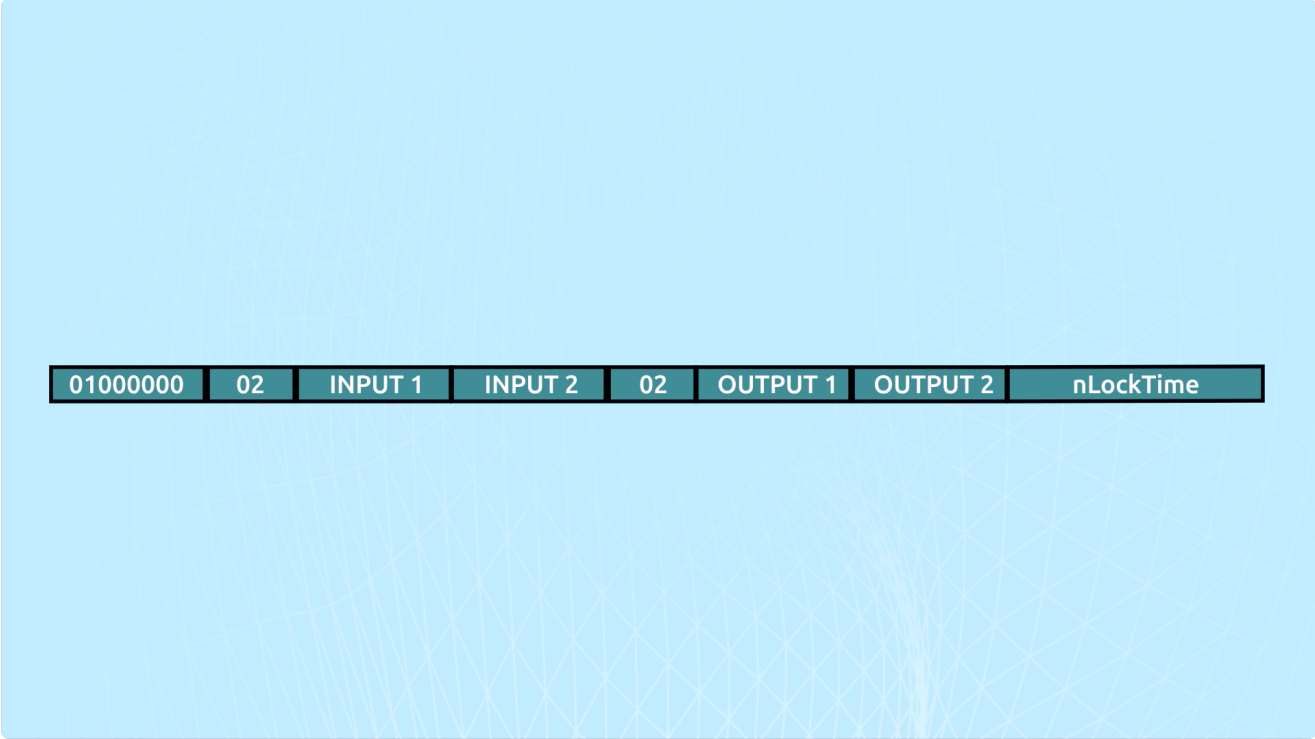
`00000000`

The final element of the transaction is its nLockTime. nLockTime is the second part of setting up payment channels.

Insert chapter 1 video 3

Payment channels are a highly useful tool and a native element of the Bitcoin protocol.

When a transaction's nLockTime is in the future, it can be considered non-final if there are also inputs with non-final nSequence values. In this particular transaction, the nLocktime is set to 0x00000000 meaning that the field references block 0 and the transaction is final at any time.



In the above animation it can be seen that the transaction is not final until either the nSequence field is UINT_MAX or the nLockTime passes which then overrides the fact that the nSequence UINT has not been incremented to its MAX value.

Chapter 1 video 3

A video on payment channels

Script	Video
Payment channels are an under-utilised feature of the Bitcoin protocol.	Todd
A payment channel is a non-final transaction. A non-final transaction can be updated to a new valid state until its locktime expires at which point it is recorded permanently on the ledger.	Use payment channel animation from Infrastruct course
Payment channels are created when a transaction is submitted with a non-final nSequence value for an input and an nLocktime in the future.	Todd
Payment channels confer immense versatility as they can be created to operate any type of program, allowing for the implementation of sophisticated functionality.	Show a computer process happening inside a transaction as the sequence number increases.
Understanding payment channels is crucial for a Bitcoin script engineer, as they serve as a valuable tool in their toolkit.	Todd

07 - The Script Evaluator

In each Bitcoin node is a system element called a script evaluator. When a Bitcoin transaction is received by a node, the evaluator performs a series of checks against it such as looking at its size, the values of its inputs and outputs and more.

During this process, the transaction is broken down into its separate component elements, such as inputs and outputs. Each part must undergo its own separate evaluation with different rules and policies.

As discussed previously, Bitcoin Scripts are processed when a transaction output is being spent as an input to a new transaction. The transaction input that spends the script must contain a valid solution to the predicate contained in the referenced output. Depending on the script used, the input can be very simple, or highly complex.

Insert new transaction validator animation (use Infrastructure one as example)

The transaction processing system must first retrieve the output being spent from the ledger. This is specified by using the TXID and index which are provided as the first part of the input. Once the output's scriptPubKey has been retrieved, the processing system appends the scriptPubKey which makes up the second part of the transaction input to the front of the scriptSig, and inserts an OP_CODESEPARATOR to demark the boundary between scriptSig and scriptPubKey. This opcode can also be used in the scriptPubKey to segregate elements of the script from parts being signed, offering novel ways to manage contracts and exchange.

The script evaluator evaluates the script from start to finish against a set of rules which are part of the Bitcoin protocol. This means that all nodes must process each script in the exact same manner for consensus to be achieved, or network forks can occur.

Transactions are only valid if all of their inputs finish processing with a single non-zero value on the stack.

Chapter 1 assessment 3

Help me Todd. Please.

Question 1

The time is 1673585961 as a UNIX epoch and the block height is 774471.

A transaction is made that spends three inputs with UINTMAX nSequences, one input with an nSequence of 0x000000FF and an nLockTime of 0x000BD1D7. Assuming regular 10 minute block intervals, after which time will the transaction be included in a block if broadcast ?

- [ ] Now
- [ ] 12 hours
- [ ] 24 hours
- [ ] 48 hours
- [ ] 72 hours
- [ ] Not until input 4's nSequence is UINTMAX.

Question 2

Flag	SIGHASH Value	Functional Meaning
SIGHASH_ALL	0x41 / 0100 0001	Sign all inputs and outputs
SIGHASH_NONE	0x42 / 0100 0010	Sign all inputs and no output
SIGHASH_SINGLE	0x43 / 0100 0011	Sign all inputs and the output with the same index
SIGHASH_ALL | ANYONECANPAY	0xC1 / 1100 0001	Sign its own input and all outputs
SIGHASH_NONE | ANYONECANPAY	0xC2 / 1100 0010	Sign its own input and no out
SIGHASH_SINGLE | ANYONECANPAY	0xC3 / 1100 0011	Sign its own input and the ou with the same index

A transaction input has the following raw hexidecimal data:
b272ec0ffa4cfd1f3446d87927004fa97177cb1569cf2008c57360a584306062320000006b483045022100c80b04357fd4daf4a0a8743bee485e40181ddc47ec81307dc1e9756ef6becc0402200d834912b5e7d071581d49373682130cd9a1f293f0552ac0194c25e7741107b04121020824bb65d1cc92de2a37410e4279211a0d53788140a528b17f7ada7f4ad8a9a9ffffffff

What is the index of the UTXO being consumed in the transaction, what is the byte length of the DER signature (excluding SIGHASH flag) and which SIGHASH flag was used?

- [ ] 32, 106 , SIGHASH_ALL
- [ ] 32, 71, SIGHASH_SINGLE
- [ ] 50, 70, SIGHASH_ALL|SIGHASH_ANYONECANPAY
- [ ] 50, 71, SIGHASH_ALL
- [ ] 50, 106, SIGHASH_NONE

Chapter 2: Basic Script Syntax

01 - Introduction

Bitcoin Script is made up 186 opcodes. Each one carries separate functionality that can be used as an instruction in the Script Evaluation Engine.

Insert Chapter 2 video 1

When a script is to be processed, the evaluation engine first loads the data from the scriptSig and processes it using the script contained in the scriptPubKey.

The processing power, or CPU needed to execute script is very low. All of the scripts fail or execute in a finite, predictable way in a limited amount of time.

As described by Satoshi in one of his earlier writings -

“The script is actually a predicate. It's just an equation that evaluates to true or false. Predicate is a long and unfamiliar word so I called it script. “ - Satoshi Nakamoto

During the validation of the transaction, the scriptSig is prefixed to the scriptPubKey from the UTXO being spent to create the full script. The full script is processed and if the final result is evaluated to be true the transaction validation is successful. We will discuss in detail how this happens later in the chapter but for now it's enough to say True is represtented by a state where a single non zero value is at the top of the stack, and False when a zero or null value is at the top of the stack. If the script processing ends with False, or with multiple items on the stack, it means that the transaction validation failed and it can not be timestamped in a block.

A good analogy for how this works is that the output scripts are puzzles that specify in which conditions can those bitcoins be spent. The input scripts provide the correct data to make those output scripts evaluate to true. In his early writings, Satoshi Nakamoto called bitcoin scripts predicates. Each output is an incomplete statement needing input data to make it true. This is the reason why processing always ends in a result which is either true or false. False outcomes fail the test and do not get published.

Chapter 2 video 1

Script	Video
In Bitcoin Script, opcodes instruct the evaluation engine to perform specific checks on a given input to determine its spendability.	Todd
The opcodes are a simple set of tools for handling items which can be used to define any type of check needed	Play a transaction validation animation
The inputs to a unlockScript can include anything from signatures to keys to arrays of data or documents.	Show different elements being inserted into unlockScript
It's the script engineer's job to create the most efficient and effective way of processing the required information for the designed application.	Todd

02 - Rules Around Data and Scripting Grammar

All data items in Bitcoin Script are a byte sequence. Some operations interpret their parameters as numeric or boolean values and require the item to fulfil the specifications of those types. Some operations produce items on the stack which are valid numeric or boolean values.

A byte sequence has a length and a value. The length of the byte sequence must be an integer greater or equal to zero and less than or equal to 2^32-1 (UINT32_MAX).

When a value is being treated as an integer, the most significant bit of the value is used to represent sign, with 1 indicating a negative value, and 0 indicating a positive value. The magnitude of its value is the same regardless of the sign bit's status.

Hexadecimal values 0x80, 0x0080, 0x00000080 are treated by arithmetic opcodes as 'negative zero'. Note the little endian notation. A script that terminates with a negative zero value on its stack will fail.

The byte sequence of length zero is called a “null item”.

Any data item can be interpreted as a boolean value. If the data item consists entirely of bytes with value zero (including negative zero), or the data item is a null item, then the boolean value of the item is false. Otherwise, the boolean value of the item is true.

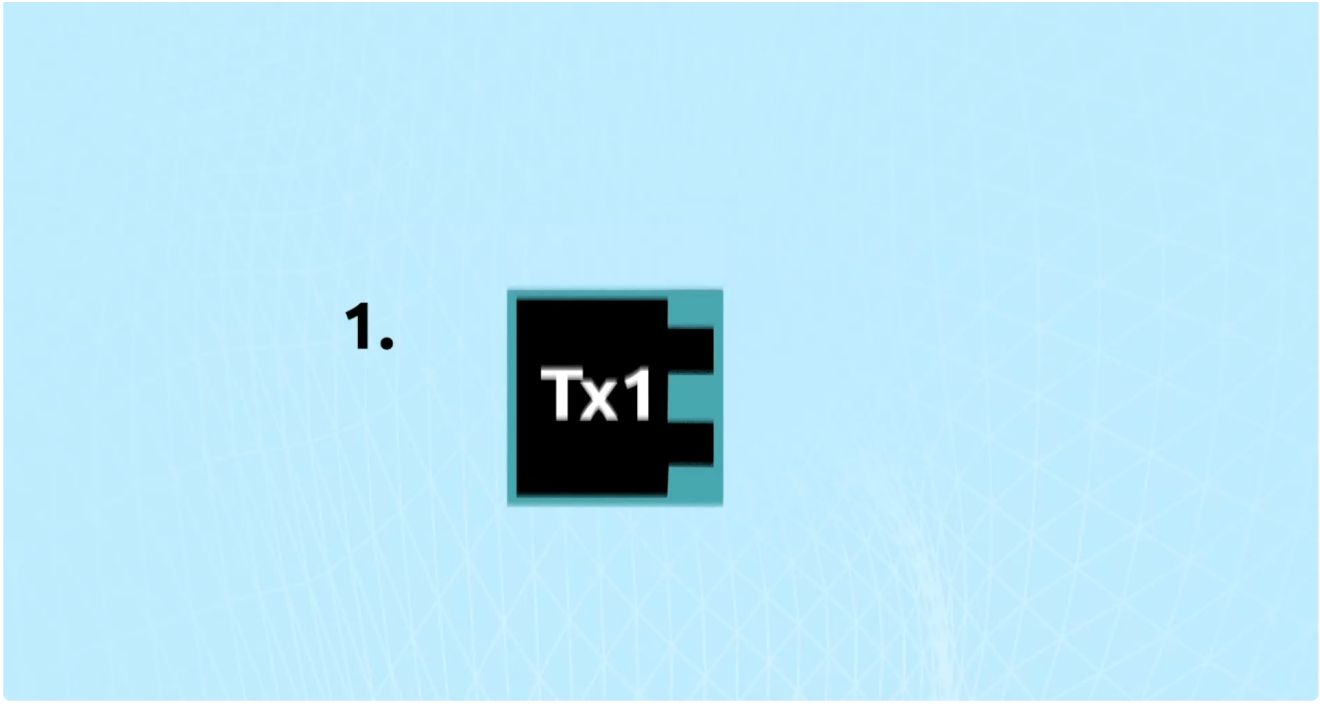
A data item can be interpreted as a numeric value. The numeric value is encoded in a byte sequence using little-endian notation. When script items are processed using opcodes that perform mathematical functions, the node will treat any byte sequence of up to 750,000 bytes length as a numeric value, allowing for 'bignum' calculations to be performed in script.

Formal Grammar for Bitcoin Script

The Formal Grammar for Bitcoin Script is defined as part of the Bitcoin protocol. This contains the full set of approved opcodes and their exact spelling and function.

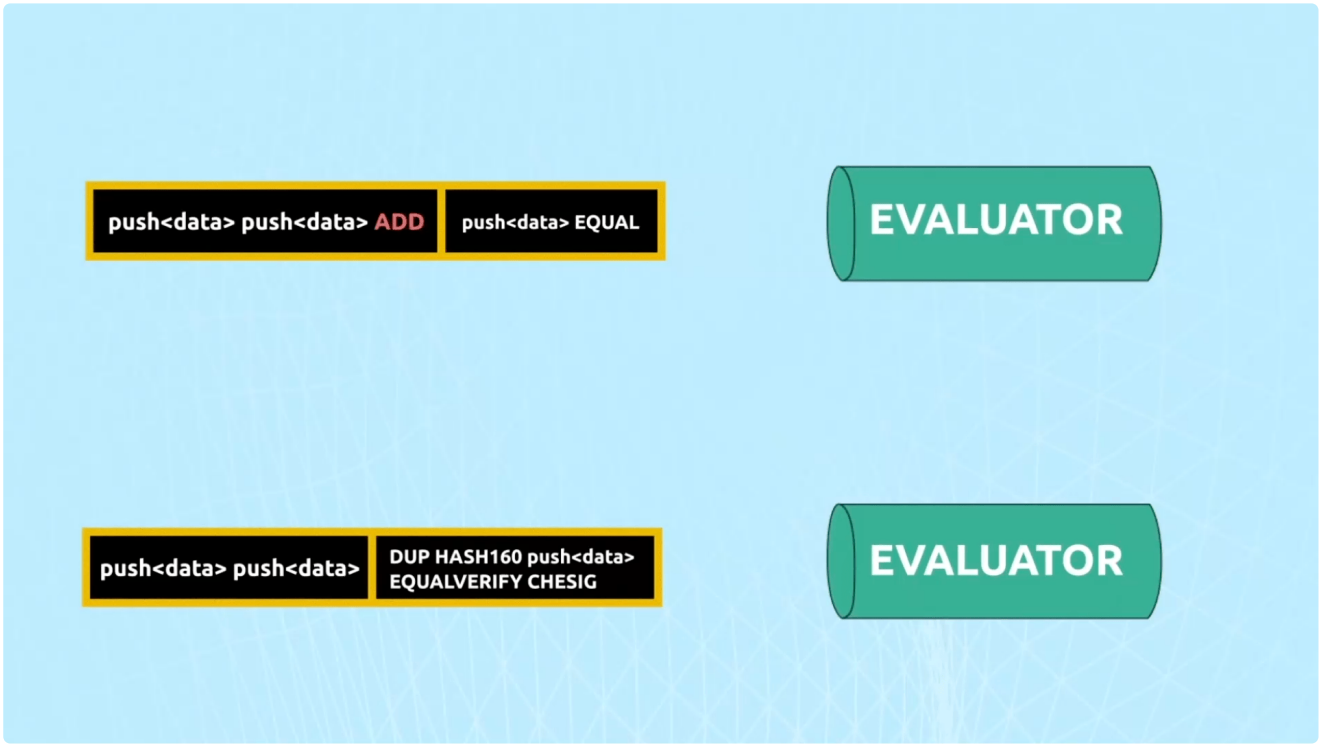
Script components

The complete script consists of two sections, the unlocking script (scriptSig) and the locking script (scriptPubKey). The locking script is from the transaction output that is being spent, while the unlocking script is included in the transaction input that is spending the output.



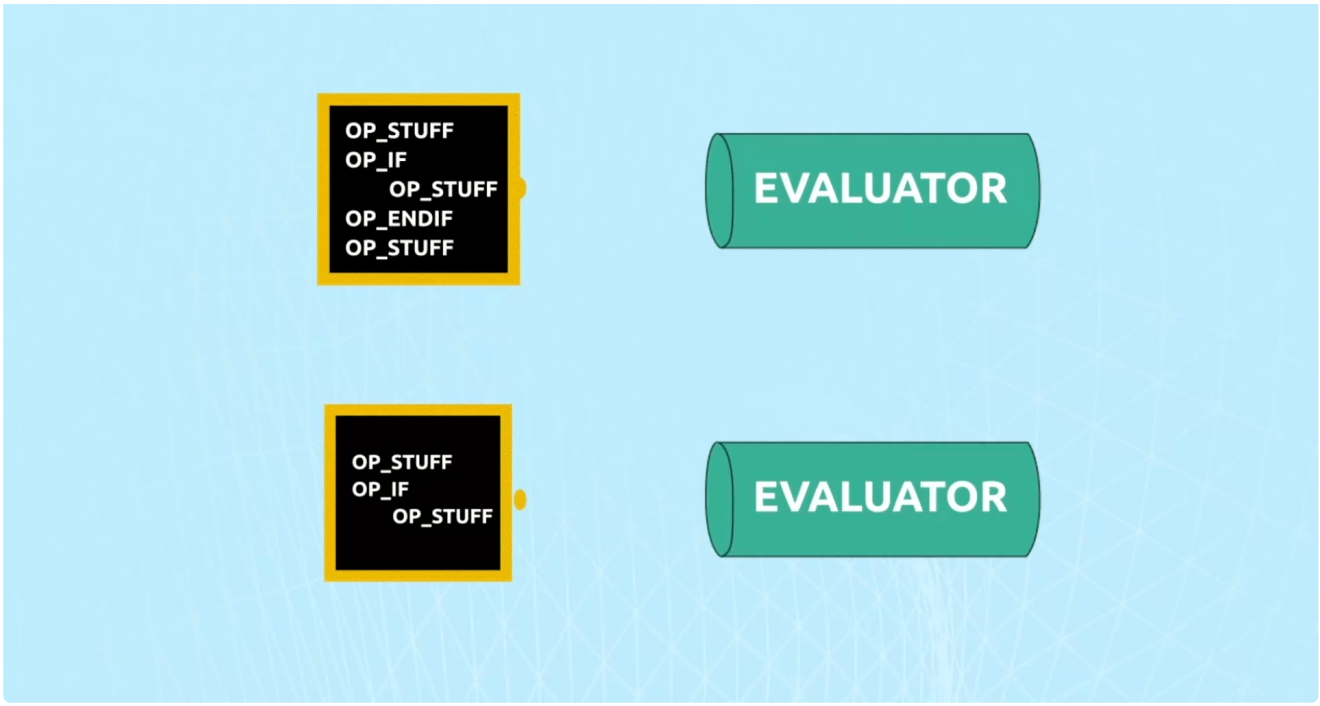
Valid opcodes for scriptSig elements

Current consensus rules state that a scriptSig (unlocking script) can only contain the first 96 opcodes, which allow constants and data to be pushed onto the stack. This requirement is a part of Validity of Script Consensus Rule, defined later.

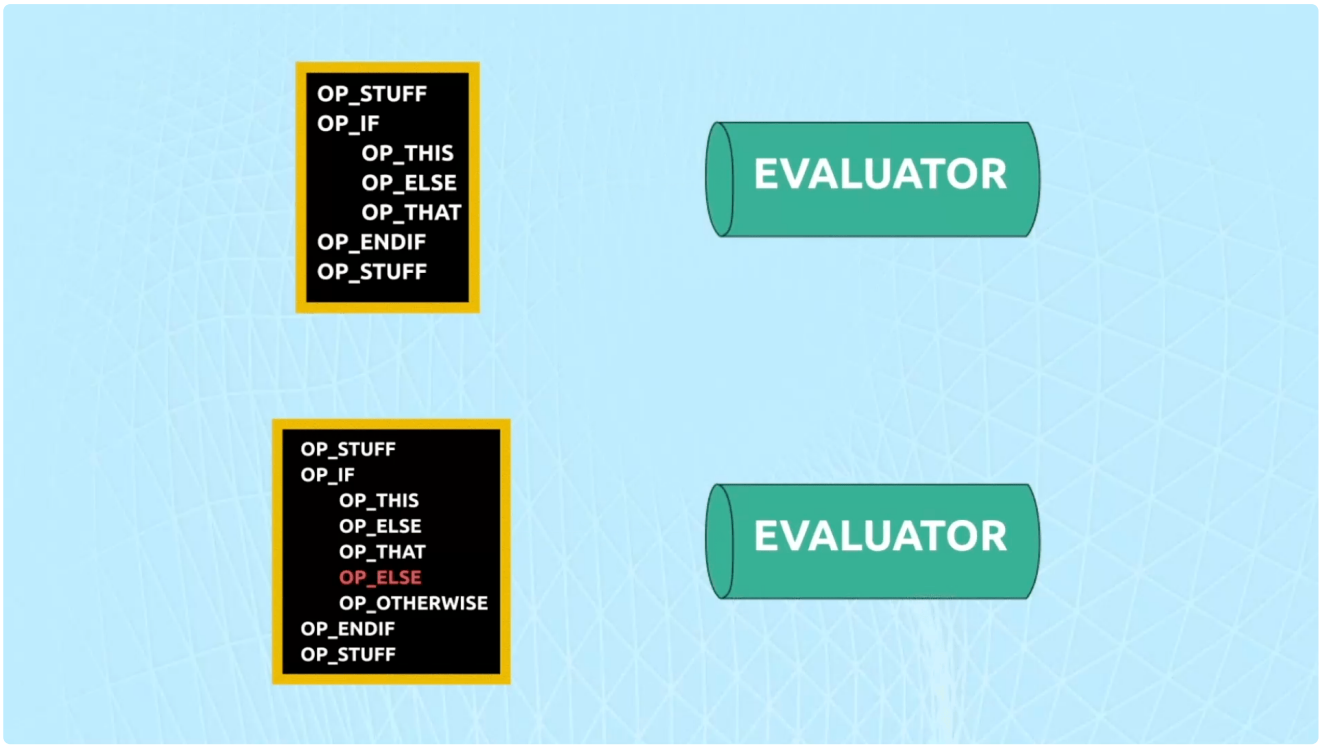


IF loops

A branching operator (OP_IF or OP_NOTIF) must have a matching OP_ENDIF.



An OP_ELSE can only be included between a branching operator and OP_ENDIF pair. There can only be at most one OP_ELSE between a branching operator and an OP_ENDIF.



OP_RETURN

OP_RETURN may appear at any location in a valid script. The functionality of OP_RETURN has been restored and is defined later in the section on opcodes. Grammatically, any bytes after an OP_RETURN that is not in a branch block are not evaluated and there are no grammatical requirements for those bytes.

Insert Chapter 2 animation 5

Note that disabled operations are part of this grammar. A disabled operation is grammatically correct but

will produce a failure if executed.

Validity of Script Rule

The inputs and outputs of a transaction must be grammatically valid, as defined by the formal grammar rules from the previous page.

The unlocking scripts used in transaction inputs may only contain PUSHDATA operations, as defined by the formal grammar above.

Currently, the following 5 opcodes are disabled: OP_2MUL, OP_2DIV, OP_VER, OP_VERIF, OP_VERNOTIF.

Numeric Value Size Rule

For a byte sequence to validly represent a numeric value, the length of the byte sequence must be less than or equal to 750,000 bytes. A byte sequence that is larger than this is a valid byte sequence but is not a valid numeric value.

This animation is INCORRECT and must be fixed both here and in Introduction to Infrastructure

Note that while some operations require parameters to be valid numeric values, they may produce byte sequences which are not valid numeric values (for example, OP_MUL may produce a byte sequence which is too large to validly represent a numeric value).

Clean Stack Rule

The Clean Stack Rule requires that a valid solution leave just one non-zero value on the stack. This means you must remove any other data items from the stack in order to be able to solve a script correctly.

The clean stack rule is a later addition to the node client system and is not a fixed element of the Bitcoin

protocol. It is possible that this rule may be removed in future.

Other rules

There are a multitude of other rules that define usage of the Bitcoin network, which are both set protocol level rules that cannot change, and flexible, consensus generated usage limits that give node operators the flexibility to create a flexible and useful environment for network users. For a deeper insight into these rules, please consider doing Introduction to Bitcoin Infrastructure.

03 - The Stacks

There are 2 stacks that are accessible to users from within Bitcoin Script programs, the main stack and alt stack. The main stack is where all opcodes take their operands from, and place the results on to (where needed). For example, the code OP_ADD performs addition of the two topmost stack items, removing them from the stack and replacing them with a single data item representing the outcome.

Insert Chapter 2 video 2 here

There are a series of opcodes that are for the direct manipulation of data items that are on the stack, allowing the programmer to duplicate, change the order of, or remove entirely data items from the main stack. These opcodes and the rest are covered in the Introduction to Bitcoin Script short course available on BitcoinSV academy now. Further information can be found at

Data items on the stack

There is no limit to how large a stack item can be within Bitcoin Script, however the evaluation of the script must still adhere to network limits such as stack usage and transaction size. There is also a limit on the size of an individual item that can be pushed to the stack in Bitcoin script which is approximately 4.3GB as defined by the OP_PUSHDATA4 opcode. It is however valid within the Bitcoin protocol to push multiple 4.3GB segments onto the stack and concatenate them to create a larger item which may be hashed to validate a timestamp. Imagination is the only boundary here.

Using the AltStack

The Altstack is a First In Last Out (FILO) stack that can be used to track useful values like counters, or a set of indexed parameters being extracted in-order. The only opcodes that interact with the altstack are OP_TOALTSTACK which moves the topmost item on the main stack to the top of the altstack, and OP_FROMALTSTACK which removes the topmost item from the Altstack and place it on the top of the main stack. There are no restrictions on usage of the Altstack outside of those imposed through rules such as the total stack memory usage rule. These rules are covered in detail with many others in Introduction to Bitcoin Infrastructure.

Clean Stack Rule

The clean stack rule asserts a requirement that upon completion of a script's processing there can only be one value remaining on the stack. If there is more than one data item remaining on the stack, the script's execution fails regardless of whether the topmost item it is a zero or non-zero value. This rule is extended to the altstack, requiring that no data be left on the altstack at the termination of a script or the execution fails.

Chapter 2 video 2

A video about the stacks in Bitcoin

Script	Video
Mastering script usage relies on proper stack utilization.	Todd's face
Bitcoin Script provides two stacks, the main stack and alt stack, facilitating the creation of complex functionalities.	Show a transaction with two stacks
During the execution of the unlockScript, the stacks efficiently store the required data elements. Any type of data can be input to a transaction, or generated on the stack, as long as it meets the rules.	Show script processing and data items being loaded onto two stacks. Show different rules inclduing stack size rule, number overflow rule a processing time rule.
A script terminates by leaving a single non-zero value on the main stack.	Show final result
Any number of items of any type can be put on either stack.	Todd

Chapter 2 assessment 1

Question 1

What is the maximum byte sequence length that can be interpreted as a valid numerical value?

- [ ] 750 bytes
- [ ] 7 500 bytes
- [ ] 75 000 bytes
- [ ] 750 000 bytes

Question 2

Which of the following big endian values will be treated as a Boolean "False"?

#	Value
A	0x000000
B	0x700000
C	0x80
D	0x7000
E	0x8000

- [ ] All of the above
- [ ] A
- [ ] C & E
- [ ] A, B & D
- [ ] A, C & E
- [ ] B & D

Chapter 3: The Opcodes

01 - Introduction

The Bitcoin script language is comprised of 186 Opcodes, which together enable a full range of functionality, allowing for an evaluation script for any digital information or conditions to be built and used to govern transactional activity on the Bitcoin network. In this chapter, we will look at all of the opcodes available and their purpose within the scripting language.

Insert Chapter 3 video 1

Each opcode is listed with a list of the input values is requires, the output it generates, and a description of the function it executes on the stack.

A handy reference table of the full list of Opcodes is available at https://wiki.bitcoinsv.io/index.php/Opcodes_used_in_Bitcoin_Script

Chapter 3 video 1

Script	Video
This chapter provides a comprehensive exploration of the complete set of opcodes forming the Bitcoin scripting language.	Todd
These opcodes will be broken down into groups to make it easier to explain their functionality.	Listing of different opcode group types, as per listing in chapter details
For each opcode, detailed explanations will be provided that show you how they use the data on the stack, and their behavior	Show animation of OP_1ADD
Having a good understanding of the full list of opcodes will be a key part of your journey to becoming a Bitcoin script engineer.	Todd

02 - Constant Value and PUSHDATA Opcodes

There are two types of opcodes that can add data to the main stack in Bitcoin script:

1. Single byte opcodes that place a constant on top of the stack
2. Pushdata opcodes that allow data of any length up to 4.3GB to be placed on top of the stack

Opcodes for pushing constants onto the stack

Each of these Opcodes allows you to use a single programmatical step cause a particular numeric value to be left on the stack. There are 18 'Constant value' opcodes available in the Bitcoin scripting language as follows:

Word	Input	Output	Description
OP_0, OP_FALSE	Nothing.	Null item	An empty array of bytes is pushed onto the stack
OP_1NEGATE	Nothing.	-1	The number -1 is pushed onto the stack.
OP_1, OP_TRUE	Nothing.	1	The number 1 (`0x01`) is pushed onto the stack.
OP_2-OP_16	Nothing.	2-16	A 1-byte integer of the value in the word name (`0x0` – `0x0F`) is pushed onto the stack.

Opcodes like these can be used to feed inputs to other functions, making it simple and easy to configure operations such as multi-signature checks for groups of 16 people or less.

Insert chapter 3 animation 1

Example:

```
OP_2 <pubkey_1> <pubkey_2> <pubkey_3> OP_3 OP_CHECKMULTISIG
```

This example script is a basic 2of3 multisignature script, and uses OP_2 and OP_3 to push integer values '2' and '3' onto the stack to instruct the validation engine as to how many valid signatures are required (2) and how many public keys are in the keypool (3).

Pushdata Opcodes

Pushdata opcodes are opcodes that push data items of a particular length onto the stack. Using these opcodes it is possible to push data items from 1 byte up to 4.3 gigabytes onto the stack.

Unlike other opcodes, these opcodes do not use any items from the stack, but rather use the following data

from the script as instructions for what to push to the stack.

Word	Input	Output	Description
Pushdata Bytelength in HEX	none	data	The next <*opcode*> bytes is data to be pushed onto the stack. Used for data items up to 75 bytes in leng
OP_PUSHDATA1	none	data	The next byte contains the number of bytes to be pushed onto the stack. Used for data items from 76 bytes to 255 bytes in length.
OP_PUSHDATA2	(special)	data	The next two bytes contain the integer number of by to be pushed onto the stack in little endian format. Used for data items from 256 bytes to 65,535 bytes length.
OP_PUSHDATA4	(special)	data	The next four bytes contain the integer number of bytes to be pushed onto the stack in little endian format. Used for data items from 65,536 bytes to 4,294,967,295 bytes in length.

Insert chapter 3 animation 2

Pushdata for items up to 75 bytes

For data items that are 75 bytes or less, the scripting language has separate opcodes that allow those items to be pushed onto the stack. These opcodes are useful for typical data items used in Bitcoin script which include public keys and ECDSA signatures in Bitcoin format.

Example:

`0x48 <signature> 0x20 <public_key> OP_CODESEPARATOR OP_DUP OP_HASH160 0x14 <public_key_hash> OP_EQUALVERIFY OP_CHECKSIG`

This example is a fully assembled 'Pay to Public Key Hash' script as it would be evaluated by a node on the network. In this example, everything after `OP_CODESEPARATOR` is retrieved from the UTXO being spent, and everything before `OP_CODESEPARATOR` is supplied by the spending party as the scriptSig. `OP_CODESEPARATOR` is inserted by the script evaluation engine to mark the separation between scriptSig and scriptPubKey.

There are three examples of pushdata opcodes in the script.

1. The first uses the `0x48` opcode to push a 72 byte signature onto the stack.
2. The second uses the `0x20` opcode to push a 32 byte public key onto the stack.
3. The third uses `0x14` to push the expected 20 byte public key hash onto the stack.

We will explore Pay to Public Key Hash (P2PKH) scripts in more detail in chapter 3.

Larger pushdata items

For larger data items, a set of three pushdata opcodes, OP_PUSHDATA1, OP_PUSHDATA2 and OP_PUSHDATA4 are available. These opcodes each expect a pair of data items to be present in the subsequent script. The first is a length indicator containing the length of the data item being added to the script. The size of this data item is dependent on the opcode used, with OP_PUSHDATA1 expecting a 1 byte length indicator used to push up to 255 bytes, OP_PUSHDATA2 expecting a 2 byte indicator used to push up to 65535 bytes and OP_PUSHDATA4 expecting a 4 byte indicator, supporting data items up to 4,294,967,296 bytes.

Example 1:

```
OP_PUSHDATA1 0x64 <100_byte_data_item>
```

This example first uses `OP_PUSHDATA1` followed by `0x64` to push a 100 byte data item to the stack

Example 2:

```
OP_PUSHDATA2 0xe803 <1kB_data_item>
```

In this example `OP_PUSHDATA2` is used followed by `0xe803` which is the little endian hexadecimal value for 1000, indicating that a 1kB data item is being pushed onto the stack.

Example 3:

```
OP_PUSHDATA4 0x40420f00 <1MB data item>
```

In this example `OP_PUSHDATA4` is used followed by `0x40420f00` which is the little endian hexadecimal value for 1,000,000, indicating that a 1MB data item is being pushed onto the stack.

Minimal Encoding Rule

One rule that is asserted by nodes on the Bitcoin network is that any pushdata operands must be 'minimally encoded'. This means that if a pushdata item of a particular length is being added to the stack, only the correct pushdata opcode can be used.

For example, to push a 100 byte data item to the stack, the following applies:

`OP_PUSHDATA1 0x64 <100B data item>` is valid Bitcoin script but `OP_PUSHDATA2 0x6400 <100B data item>` violates the minimal encoding rules and any transaction containing this script element in an output will be rejected.

Pushdata Opcode Notation In Script

It is important to note that most Bitcoin script interpreters or programming tools will insert the correct pushdata opcode for the data item being pushed, respecting the minimal encoding rule. Typically the user will need only to provide the data item itself.

Examples shown in subsequent pages/chapters will exclude pushdata opcodes from the script to simplify

the expressions and allow you to focus on the opcodes being discuss

Input Script Opcode Rule

Another rule asserted by nodes on the Bitcoin network is that only constant data and pushdata opcodes can be used to form valid input scripts. Any transaction submitted to the network that uses opcodes other than constant data/pushdata opcodes is considered invalid and will be rejected.

Chapter 3 animation 1

Show a script as follows:

OP_1 OP_2 OP_3 OP_4 OP_5 OP_6 OP_7 OP_8 OP_9 OP_10 OP_11 OP_12 OP_13 OP_14 OP_15 OP_16

Show an empty stack (can be represented by an open top rectangle with depth for 16 items

Show OP_1 being used

Show 0x01 appear on the bottom of the stack

show OP_2 being used

Show 0x02 appear on the bottom of the stack

etc.

Chapter 3 animation 2

Show a script as follows:

0x14 <20 byte data item> 0x20 <32 byte data item> OP_PUSHDATA1 0x96 <150 byte data item> OP_PUSHDATA2 0x45c1 <18kB byte data item> OP_PUSHDATA4 0x004ab204 <4.9MB data item>

Show an empty stack (can be represented by an open top rectangle with depth for 16 items

Show 0x14 being used, and <20 byte data item> moving to the stack

Show OP_PUSHDATA1 and 0x96 being used and <18kB data item> moving to stack

wetc.

Chapter 3 assessment 1

1. What OPCODE would you use to push the data item 0x0129a0 onto the stack?
    1. `OP_6`
    2. `OP_3`
    3. `0x03` <-
    4. `OP_PUSHDATA4`
2. Which script correctly pushes 1, 17, 64, 256, 65536 onto the stack? (values are represented as little-endian hexadecimal numbers)
    1. `OP_1 OP_17 OP_PUSHDATA1 0x40 OP_PUSHDATA_2 0x0001 OP_PUSHDATA4 0x00000100`
    2. `0x01 0x17 0x64 OP_PUSHDATA2 0x0001 OP_PUSHDATA4 0x00000100`
    3. `OP_1 0x01 0x11 0x01 0x40 0x02 0x0001 0x03 0x000001` <-
    4. `OP_1 OP_PUSHDATA1 0x01 0x11 OP_PUSHDATA1 0x01 0x40 OP_PUSHDATA1 0x03 0x000001`
3. Which opcode would you use to push an 82,492 byte long data item to the stack?
    1. `OP_3`
    2. `OP_PUSHDATA2`
    3. `OP_PUSHDATA3`
    4. `OP_PUSHDATA4` <-

03 - IF Loops

Flow control opcodes are opcodes that allow a script to either execute or ignore particular sections of code, or to terminate the script process depending on the value of the topmost stack items.

There are a variety of OPCODES available to trigger entry into IF loops including:

`OP_IF, OP_NOTIF,`

Insert Chapter 3 video 2

IF / NOTIF statements

IF statements are used to allow Bitcoin script to do comparative analysis on stack data items to control entry into IF loops. IF loops are the only means provided in Bitcoin script to perform different operations depending on previous processing.

The format of a simple IF loop is as follows:

`<Expression>`

`OP_IF`

`    <True action>`

`OP_ENDIF`

`Insert animation of IF loop`

In this example, if the operations performed in `<expression>` leave a non-zero item at the top of the stack, then the code in `<True action>` will be executed. Otherwise the script will jump to the opcode immediately after `OP_ENDIF`.

The alternative to `OP_IF` is `OP_NOTIF`.

`<Expression>`

`OP_NOTIF`

`    <False action>`

`OP_ENDIF`

`Insert animation of NOTIF loop`

When using `OP_NOTIF`, if the operations performed in `<expression>` leave a zero-value item at the top of the stack, then the code in `<False action>` will be executed. Otherwise the script will jump to the opcode immediately after `OP_ENDIF`.

Bitcoin grammar rules require that every `OP_IF / OP_NOTIF` must have a corresponding `OP_ENDIF`. Transactions that try to create outputs that do not adhere to this scripting rule are considered invalid and will not be accepted by the network.

Using OP_ELSE

Any IF loop can contain an ELSE statement which will cause the script to branch into one of two paths depending on the outcome of the expression being evaluated.

```
<Expression>
OP_IF
    <True action>
OP_ELSE
    <False action>
OP_ENDIF
```

Multi-condition loops / Case statements

Using OP_ELSE, IF loops can be nested allowing for complex nested functions to be developed. This allows for similar functionality to a case statement to be implemented.

```
<Case 1 check>
OP_IF
    <When 1 action>
OP_ELSE
    <Case 2 check>
    OP_IF
        <When 2 action>
    OP_ELSE
        <Else action>
    OP_ENDIF
OP_ENDIF
```

`Insert animation of IF/ELSE loop`

An alternative method to nested IF loops is repeating IF loops in the code, although care must be taken to ensure that any 'ELSE' case is captured in an IF loop separately.

```
<Case 1 check>
OP_IF
    <When 1 action>
OP_ENDIF
<Case 2 check>
OP_IF
    <When 2 action>
OP_ENDIF
<else check>
OP_IF
    <Else action>
OP_ENDIF
```

Animation of IF loop

Animation of NOTIF loop

Animation of IF/ELSE loop

Chapter 3 video 2

If loops

If loops are a vital part of Bitcoin script, allowing you to create branches in code that	Todd
process alternative elements of the total predicate dependent on certain input conditions.	Show Animation of IF loop
There are two opcodes that can trigger entry into an IF loop. These are:	Todd
OP_IF and OP_NOTIF	Show words on screen
IF loops can also contain an 'ELSE branch' where the alternative outcome to the tested outcome is processed. IF loops can be nested to test multiple outcomes, for the creation of scripts enabling the emulation of case statements and ELSE-IF functionality.	Show Animation of IF/ELSE loop
These opcodes can be used to develop advanced functionality and are an essential part of the Bitcoin script engineer's toolbox.	Todd

Chapter 3 assessment 2

Which script is invalid?

1. `<input> OP_IF OP_2 OP_ELSE OP_1 OP_ENDIF`
2. `<input_1> <input_2> OP_NOTIF 0x98 OP_EQUAL OP_IF <winner> OP_ELSE <no win> OP_ENDIF OP_ELSE <winner> OP_ENDIF`
3. `<input_1> <input_2> OP_IF OP_2 OP_IF OP_4 OP_ENDIF` <-
4. `<input_1> OP_IF OP_1 OP_ELSE OP_0 OP_ENDIF`

Which script is valid if the scriptSig is `0x00`

1. `OP_NOTIF OP_TRUE OP_IF OP_FALSE OP_ENDIF OP_ENDIF`
2. `OP_IF OP_FALSE OP_ENDIF`
3. `OP_NOTIF OP_TRUE OP_ENDIF` <-
4. `OP_NOTIF OP_FALSE OP_ELSE OP_TRUE OP_ENDIF`

04 - OP_NOP, OP_VERIFY and its Derivatives

Aside from OP_IF and its companion opcodes, there are other useful opcodes that allow us to control the sections of script being processed or not processed given the set of conditions submitted by the spending party.

OP_NOP

`OP_NOP` simply performs *no action*. It consumes nothing from the stack and leaves nothing on the stack. It can be used in sophisticated transaction templates to pad elements of a new script that does not match a size requirement.

Insert OP_NOP from Chapter3 animation pack 1

OP_VERIFY

`OP_VERIFY` acts as a gating function against any condition being evaluated. OP_VERIFY consumes the top value on the stack. If the value is any non-zero value, the opcode allows the script to continue or if it is a zero value item, causes it to terminate and fail.

Insert OP_VERIFY from Chapter3 animation pack 1

The 'Verify' functionality is also added to the functionality of several available opcodes to create more efficient versions of specific functions as outlined below:

OP_EQUALVERIFY

`OP_EQUALVERIFY` only allows the script to continue processing if the top two data items on the stack are identical bytevectors. If the data items are not equal, the opcode causes the script to terminate and fail. The opcode performs the same function produced by `OP_EQUAL OP_VERIFY`.

Insert OP_EQUALVERIFY from Chapter3 animation pack 1

Example:

```
OP_2 OP_ADD OP_3 OP_EQUALVERIFY
```

In this example, if the integer value 1 is on the top of the stack when this snippet begins processing, the script will continue to process beyond OP_EQUALVERIFY. If any other value is on top of the stack, the script will fail.

OP_NUMEQUALVERIFY

`OP_NUMEQUALVERIFY` only allows the script to continue processing if the top two data items on the

stack have identical integer values. If the data items are not numerically equal, the opcode causes the script to terminate and fail. This can be useful when values are being calculated which may finish with uncertain bytevector lengths.

Insert OP_NUMEQUALVERIFY from Chapter3 animation pack 1

Example:

```
OP_4 OP_SPLIT OP_DROP OP_1 OP_NUMEQUALVERIFY
```

In this example, a value of any length can be put onto the stack. The script splits the first 8 characters (4 bytes) from the value and drops the remainder. It then checks that it is numerically equal to 1. Bitcoin is little endian so this would mean the string's first 4 bytes are 01000000. If the value is numerically equal to 1, the script will continue processing while any other value will result in failure. This is useful to keep scripts compact, or when dealing user inputs of uncertain length.

OP_CHECKSIGVERIFY

OP_CHECKSIGVERIFY performs an ECDSA signature check consuming the top two values on the stack and allows the script to continue processing if the signature is valid.

Insert OP_CHECKSIGVERIFY from Chapter3 animation pack 1

Example:

```
<signature> <public_key> OP_CHECKSIGVERIFY
```

If the signature is invalid, the `OP_CHECKSIGVERIFY` opcode causes the script to terminate and fail.

This is a basic signature check script, and is occasionally used by transactions on the network, although most wallets favor Pay To Public Key Hash (P2PKH) as the default script format.

OP_CHECKMULTISIGVERIFY

OP_CHECKMULTISIGVERIFY performs an ECDSA signature check consuming up to i items from the stack, where n is the number of public keys in the group, m is the number of signatures needed and i = n + m + 3. If the multisignature condition is valid, the script continues to process.

Insert OP_CHECKMULTISIGVERIFY from Chapter3 animation pack 1

Example:

```
OP_1* <signature_1> <signature_3> OP_2 <public_key 1> <public_key 2>
<public_key 3> OP_3 OP_CHECKMULTISIGVERIFY
```

*Note that the first item pushed using OP_1 is called the junk item. This will be covered in detail in chapter 5

In this example, n = 2 and m = 3 so there are 2(signatures) + 3(pubkeys) + 2(indicators) + 1(junk item) = 8

items consumed from the stack. To spend an output with this type of multisignature script, the user must submit a junk item and 2 valid signatures corresponding to 2 of the public keys in the list, in the correct order.

If the multisignature check is invalid, the opcode causes the script to terminate and fail.

Chapter 3 animation pack 1

Show scripts as follows:

`OP_NOP` show that nothing happens

`<input> OP_VERIFY` show that the script continues to process when input is non-zero and fails when it is zero

`<input_1> <input_2> OP_EQUALVERIFY` show that the script continues to process when inputs are exactly equal and fails when they are not equal. Use the following examples

0x01 and 0x01 EQUAL

0x01 and 0x02 NOT EQUAL

0x01 and 0x0001 NOT EQUAL

`<input> OP_NUMEQUALVERIFY` show that the script continues to process when inputs are numerically equal and fails when they are not numerically equal. Use the following examples

0x01 and 0x01 EQUAL

0x01 and 0x02 NOT EQUAL

0x01 and 0x0001 EQUAL

`<signature> <public key OP_CHECKSIGVERIFY` Show the sgnature being checked against the public key and the script either failing or continuing to process.

`OP_1 <signature_1> <signature_2> OP_2 <pubkey_1> <pubkey_2> <pubkey_3> OP_3 OP_CHECKMULTISIGVERIFY` Show all stack items being consumed by opcode. and the script either failing or continuing to process.

05 - OP_RETURN

OP_RETURN has a storied history which will not be covered in this content. More information can be found here.

OP_RETURN causes the script to terminate. If there is a single non-zero item on the stack the script will return TRUE, allowing it to be spent. If the topmost stack value is zero, or there are multiple items on the stack, the script fails and the transaction cannot be submitted to the blockchain.

This functionality can be applied to a multitude of use cases.

Insert chapter 3 video 3

Example:

```
OP_DEPTH OP_1 OP_EQUAL OP_IF
    <public_key> OP_CHECKSIG
    OP_RETURN
OP_ENDIF
<rest_of_script>
```

In this example we first check the depth of the stack. This tells us how many items there are on the stack at this moment in the script processing. If there is just 1 item, the script expects that this will be a signature, that can be verified using the public key in the script. If the OP_CHECKSIG operation finds a valid signature, OP_RETURN will end the script successfully. If the signature check is invalid, the script will terminate in failure, and the output cannot be spent.

In this example, it is assumed that any further actions that take place in <rest_of_script> require two or more stack items in order to successfully validate.

The functionality of OP_CHECKSIG and other variations is covered in page 9 of this chapter.

FALSE RETURN scripts

One particular use-case for OP_RETURN is in the creation of FALSE RETURN scripts (commonly called 'OP_RETURN outputs').

A False Return script is created by generating a transaction output which uses OP_FALSE and OP_RETURN as the first two opcodes in its script. Because a script that begins with OP_FALSE OP_RETURN will always terminate in failure, these scripts are considered unspendable and are the only type of script that can be published with an output value of zero. As such they can be used as carriers for data items that do not form part of any locking conditions.

This technique is widely used on the BitcoinSV network for a variety of token solutions, and to allow platforms and applications to capture data onto the blockchain for indexing and analysis.

Example:

```
OP_FALSE OP_RETURN <data packet>
```

In this example, `OP_FALSE` is pushed onto the stack before the `OP_RETURN` opcode ends the script. Because OP_RETURN always returns with a false result this script can never be spent.

Chapter 3 video 3

OP_RETURN features

Script	Video
OP_RETURN is a highly utile opcode that is available within Bitcoin's scripting language.	Todd
When a script encounters an OP_RETURN opcode, it immediately terminates. This feature proves valuable as it enables scripts to exit early using an administrative key or terminate prematurely upon identifying invalid conditions.	Show a script processing and reaching an OP_RETURN then terminating. Script details: <signature> <pubkey> <0> || OP_NOTIF OP_DU OP_HASH160 <hash> OP_EQUAL OP_CHECKSIG OP_RETURN OP_ELSE 2 <pubkey1> <pubkey2> <pubkey3> 3 OP_CHECKMULTISIG OP_ENDIF
For several years OP_RETURN's functionality was limited such that any script that used it would be rendered unspendable.	Todd
This functionality was still useful as it gave developers a way to create blockchain records that carried no satoshi value	Show transaction with 2 outputs, one with satos and one with a FALSE RETURN output
but which could contain arbitrary data items.	Todd
With the functionality of OP_RETURN being reverted, a new requirement emerged where scripts utilizing it must begin with OP_FALSE OP_RETURN.	Show OP_FALSE OP_RETURN with data item
These 'false return' outputs are still very widely used in Bitcoin and will remain an important utility function available to all creators.	Todd

06 - Stack Operations

Stack operations include opcodes who's primary purpose is to process modifications to the stack, without creating any new or modified values.

Insert Chapter 3 video 4

Stack Duplicators

Stack duplicators are opcodes that duplicate items at one or more locations on the stack. The Duplicators are as follows:

Word	Input	Output	Description
OP_DUP	x	x x	Duplicates the top stack item.
OP_2DUP	x1 x2	x1 x2 x1 x2	Duplicates the top two stack items.
OP_3DUP	x1 x2 x3	x1 x2 x3 x1 x2 x3	Duplicates the top three stack items
OP_IFDUP	x	x / x x	If the top stack value is not 0, duplic it.
OP_OVER	x1 x2	x1 x2 x1	Copies the second-to-top stack item the top.
OP_2OVER	x1 x2 x3 x4	x1 x2 x3 x4 x1 x2	Copies the pair of items two spaces back in the stack to the front.
OP_PICK	xn ... x2 x1 x0 <n>	xn ... x2 x1 x0 xn	The item *n* back in the stack is copie to the top.
OP_TUCK	x1 x2	x2 x1 x2	The item at the top of the stack is copied and inserted before the seco to-top item.

Insert Chapter 3 animation pack 4

Stack Eliminators

Stack eliminators are opcodes that eliminate one or more items from the stack. The eliminators are as follows:

Word	Input	Output	Description
OP_DROP	x	Nothing	Removes the top stack item.
OP_2DROP	x1 x2	Nothing	Removes the top two stack items.
OP_NIP	x1 x2	x2	Removes the second-to-top stack item.

Insert Chapter 3 animation pack 5

Stack Relocators

Stack relocators are opcodes that move one or more items from their locations on the stack to a new location on the stack. The relocators are as follows:

Word	Input	Output	Description
OP_SWAP	x1 x2	x2 x1	The top two items on the stack are swapped.
OP_2SWAP	x1 x2 x3 x4	x3 x4 x1 x2	Swaps the top two pairs of items.
OP_ROT	x1 x2 x3	x2 x3 x1	The top three items on the stack are rotated to the left.
OP_2ROT	x1 x2 x3 x4 x5 x6	x3 x4 x5 x6 x1 x2	The fifth and sixth items back are moved to the top of the stack.
OP_ROLL	xn ... x2 x1 x0 <n>	... x2 x1 x0 xn	The item *n* back in the stack is mov to the top.

Insert Chapter 3 animation pack 6

Alt Stack Operators

The Bitcoin Script evaluation engine has a secondary 'Altstack' available to store data as needed for calculations. There are just 2 opcodes used to interface with the altstack, one which pushes data items onto the top and one which pulls data items from the top. The Altstack operates as a LIFO (Last In First Out) buffer.

The altstack is useful for storing items for later processing that would otherwise have to be moved to the back of the stack and retrieved,

Word	Input	Output	Description
OP_TOALTSTACK	x1	(alt)x1	Puts the input onto the top of the alt stack. Removes from the main stack.
OP_FROMALTSTACK	(alt)x1	x1	Puts the input onto the top of the main stack. Remov it from the alt stack.

Insert Chapter 3 animation pack 7

Example 1:

```
OP_DUP OP_1 OP_EQUAL OP_IF
    OP_DROP <pubkey1> OP_CHECKSIG
OP_ELSE
    OP_2 OP_EQUAL OP_IF
        <pubkey2> OP_CHECKSIG
    OP_ELSE
        OP_FALSE OP_RETURN
    OP_ENDIF
OP_ENDIF
```

In this example, one of two parties can sign a transaction, and indicate which party is signing the script by appending a 1 or a 2 to their solution. This can be spent using one of the two following scriptSigs:

```
<signature1> <1>
<signature2> <2>
```

If any other value other than 1 or 2 is at the top of the stack, the script will fail.

This can be useful as it forces the signing party to indicate which key is being used within the script, allowing their identity to be captured without evaluating signatures.

Example 2:

```
OP_2DROP <pubkey> OP_CHECKSIG
```

In this example, the output being spent is used to capture data on the ledger which is not relevant to the spending of the token. This script can be validly spent using the following scriptSig:

```
<signature> <data_item1> <data_item2>
```

Example 3:

```
OP_IFDUP OP_IF
    <pubkey1> <pubkey2> 2 OP_CHECKMULTISIG
OP_ELSE
   <pubkey> OP_CHECKSIG
OP_ENDIF
```

In this example, the first loop can be entered if either 1, 2, or 3 of the signing parties required choose to sign using one of the following scriptSigs:

```
<x> <signature1> <1>
<x> <signature2> <1>
<x> <signature1> <signature2> <2>
```

In these solutions, the number of signatures to be checked is the last item on the stack, confirming the number of signatures needed.

To enter the second loop, the signing party would use the following scriptSig:

```
<signature> <0>
```

The 0 tells the script that the second entity wishes to sign using a single signature check, so there is no need to keep the value on the stack.

Example 4:

```
OP_OVER OP_SIZE OP_1SUB OP_SPLIT OP_NIP <sighash> OP_EQUAL OP_NOTIF
    OP_FALSE OP_RETURN
OP_ENDIF
```

In this example `OP_OVER` is used to bring the signature to the top of the stack. It's size is queried and subtracted by 1 before it is split, and the main part nipped from the stack, leaving just the SIGHASH flag which is then checked against a particular mask. If the correct SIGHASH flag is not used, the script fails. Opcodes such as `OP_SIZE`, `OP_EQUAL` and `OP_1SUB` will be covered in later parts of this chapter. For further information on the structure of DER signatures in Bitcoin, please visit this page.

Example 5:

```
OP_1SUB OP_DUP OP_TOALTSTACK OP_NOTIF
    2 <pubkey1> <pubkey2> 2 OP_CHECKMULTISIG
    OP_FROMALTSTACK OP_DROP
```

```
OP_ELSE

    OP_FROMALTSTACK OP_1SUB OP_NOTIF

        <pubkey> OP_CHECKSIG

    OP_ELSE

        OP_FALSE OP_RETURN

    OP_ENDIF

OP_ENDIF
```

In this example, the spending party is one of two entities. To spend the transaction they must indicate to the script using an integer value of 1 or 2 which path they wish to take. The first entity requires a 2of2 multisignature solution which they would solve with the following scriptSig:

```
<x> <signature1> <signature2> <1>
```

with the first element <x> being required due to a protocol bug (explained later), then the 2 signatures and the integer value <1> to show they wish to enter the first loop of the transaction.

By subtracting 1 from the value before duplicating it and storing the copy on the altstack, we can save space by performing a NOTIF check rather than comparing the value using `OP_1 OP_EQUAL OP_IF`. The value stored on the altstack is then pulled and dropped

If the second entity wishes to sign, they would use the following scriptSig:

```
<signature> <2>
```

After the loop identifier is pulled from the altstack, a second subtraction is performed before a NOTIF check is done. In the second loop a single signature check is performed before the nested IF loops are exited.

If the top value on the stack is neither 1 nor 2, the script will enter the nested `OP_ELSE` statement where `OP_FALSE OP_RETURN` will cause it to fail.

Chapter 3 video 4

Stack operations

Script	Video
Bitcoin script is a very simple programming language, but with a distinct quirk in its nature that makes it quite different to other more familiar languages out there.	Todd
Bitcoin script uses reverse polish notation and a pair of stacks to manage all data being processed in a script	Show animation of script processing and data items being added to main stack, sent to alt-stac and moved back.
The stacks are known as the Main stack and the Alt stack.	Todd
The main stack is where all opcodes in the programming language do their work.	Show opcodes changing main stack
The ALT stack is simply a Last In First Out buffer that is accessible via the opcodes OP_TOALTSTACK and OP_FROMALTSTACK.	Todd
OP_TOALTSTACK takes the topmost data item from the main stack and places it at the top of the alstack.	Show OP_TOALSTACK animation
OP_FROMALTSTACK takes the topmost data item from the altstack and places it at the top of the main stack	Show OP_FROMALTSTACK animation
The stack operations in Bitcoin script provide essential functionality for managing and manipulating data during script execution.	Todd
These operations ensure the proper flow of data between the Main stack and the Alt stack.	
This feature empowers developers to process data effectively and adapt script behavior based on specific requirements.	Todd

Chapter 3 animation pack 4

Animate the following functions:

Word	Input	Output	Description
OP_DUP	x	x x	Duplicates the top stack item.
OP_2DUP	x1 x2	x1 x2 x1 x2	Duplicates the top two stack items.
OP_3DUP	x1 x2 x3	x1 x2 x3 x1 x2 x3	Duplicates the top three stack items
OP_IFDUP	x	x / x x	If the top stack value is not 0, duplic it.
OP_OVER	x1 x2	x1 x2 x1	Copies the second-to-top stack item the top.
OP_2OVER	x1 x2 x3 x4	x1 x2 x3 x4 x1 x2	Copies the pair of items two spaces back in the stack to the front.
OP_PICK	xn ... x2 x1 x0 <n>	xn ... x2 x1 x0 xn	The item *n* back in the stack is copie to the top.
OP_TUCK	x1 x2	x2 x1 x2	The item at the top of the stack is copied and inserted before the seco to-top item.

Chapter 3 animation pack 5

Animate the following functions:

Word	Input	Output	Description
OP_DROP	x	Nothing	Removes the top stack item.
OP_2DROP	x1 x2	Nothing	Removes the top two stack items.
OP_NIP	x1 x2	x2	Removes the second-to-top stack item.

Chapter 3 animation pack 6

Animate the following functions:

Word	Input	Output	Description
OP_SWAP	x1 x2	x2 x1	The top two items on the stack are swapped.
OP_2SWAP	x1 x2 x3 x4	x3 x4 x1 x2	Swaps the top two pairs of items.
OP_ROT	x1 x2 x3	x2 x3 x1	The top three items on the stack are rotated to the left.
OP_2ROT	x1 x2 x3 x4 x5 x6	x3 x4 x5 x6 x1 x2	The fifth and sixth items back are moved to the top of the stack.
OP_ROLL	xn ... x2 x1 x0 <n>	... x2 x1 x0 xn	The item *n* back in the stack is mov to the top.

Chapter 3 animation pack 7

Animate the following functions:

Word	Input	Output	Description
OP_TOALTSTACK	x1	(alt)x1	Puts the input onto the top of the alt stack. Removes from the main stack.
OP_FROMALTSTACK	(alt)x1	x1	Puts the input onto the top of the main stack. Remov it from the alt stack.

Chapter 3 assessment 3

1. Which script returns true if <input> is either 0x00 or 0x01?
    1. `<input> OP_DUP OP_IF OP_1 OP_ELSE OP_FALSE OP_ENDIF OP_NUMEQUALVERIFY`
    2. `<input> OP_IF OP_1 OP_NUMEQUALVERIFY OP_ENDIF`
    3. `<input> OP_DUP OP_NOTIF OP_RETURN OP_ELSE OP_1 OP_NUMEQUALVERIFY OP_ENDIF`
    4. `<input> OP_IFDUP OP_IF OP_1 OP_EQUALVERIFY OP_ENDIF OP_1` <-

2. Which script fails if <input0> <input1> and <input2> are 0, 1 and 2 respectively?
    1. `<input0> <input1> <input2> OP_0 OP_EQUALVERIFY OP_1 OP_EQUALVERIFY OP_2 OP_EQUALVERIFY` <-
    2. `<input0> <input1> <input2> OP_2 OP_EQUALVERIFY OP_1 OP_EQUALVERIFY OP_FALSE OP_EQUALVERIFY OP_TRUE`
    3. `<input0> <input1> <input2> OP_2 OP_ROLL OP_NOTIF OP_SWAP OP_1 OP_EQUALVERIFY OP_2 OP_EQUALVERIFY OP_1 OP_ENDIF`
    4. `<input0> <input1> <input2> OP_SWAP OP_PICK OP_PICK`

07 - Data transformation

Data transformation opcodes are used to change the data on the stack, either by splitting one item into multiple items, joining items together, or changing an item to a forms required by the script processing engine.

Splitting and joining data items

When handling data items on the stack, they may need to be split or joined as part of the script functionality. There are two opcodes for these functions as defined below:

Word	Input	Output	Description
OP_CAT	x1 x2	out	Concatenates two strings.
OP_SPLIT	x n	x1 x2	Splits byte sequence x at position n.

Insert Chapter 3 animation 8

Example:

```
0x12345678 OP_1 OP_SPLIT
```

This example splits the bytevector `0x12345678` into `0x12` and `0x345678`

Example:

```
0x12345678 OP_1 OP_SPLIT OP_1 OP_SPLIT OP_1 OP_SPLIT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
```

This example snippet changes the endianness of the 4 byte integer 0x12345678, leaving 0x78563412 on the stack. This is useful when transaction inputs are submitted in big-endian format but need to be changed to little endian to undergo mathematical transformation.

Data Type Transformations

These two opcodes are not part of the original Bitcoin script definition. They were used when Bitcoin script limited mathematical functions to data items no larger than 4 bytes in size. OP_NUM2BIN can be used when string items need to be extended to arbitrary lengths with leading zeroes. OP_BIN2NUM transforms a string to a minimally encoded number by stripping any leading zeroes.

Word	Input	Output	Description
OP_NUM2BIN	a b	out	Converts numeric value a into byte sequence of length b.
OP_BIN2NUM	x	out	Converts byte sequence x into a minimally encoded num value.

Insert Chapter 3 video 5

Example 1:

`OP_16 OP_NUM2BIN`

In this example, a string of uncertain length is extended to 16 bytes.

e.g. `0x0102030405060708` -> `0x01020304050607080900000000000000`

Example 2:

`OP_BIN2NUM`

In this example, a string of uncertain length is cut back to its minimal integer representation

e.g. `0x01020304050607080900000000000000` -> `0x0102030405060708`

Chapter 3 animation pack 8

Animate the following functions:

Word	Input	Output	Description
OP_CAT	x1 x2	out	Concatenates two strings.
OP_SPLIT	x n	x1 x2	Splits byte sequence x at position n.

Chapter 3 video 5

Data type transformation explainer

Script	Video
While the nodes on the BSV network have consistently advocated for no changes to the underlying Bitcoin protocol, there have been instances where this principle has been deviated from, and the following two opcodes serve as evident examples.	Todd
OP_NUM2BIN and OP_BIN2NUM, which replaced other opcodes, diverge from the original Bitcoin script and remain as the only active opcodes today that were not part of the originally released protocol.	Show wiki opcode page and highlight NUM2BIN and BIN2NUM https://wiki.bitcoinsv.io/index.php/Opcodes_used_in_Bitcoin_Script#Data_Manipulation
OP_NUM2BIN is utilized to pad numeric values to a specified length, commonly used when a mask of a particular size is required. However, it's important to exercise caution as this opcode zeroes the sign bit, potentially altering the original value.	Show a number being extended to match a mas also show a negative number being changed to positive and extended
OP_BIN2NUM removes leading zeroes from a string, resulting in a minimally encoded numeric value placed at the top of the stack.	Show an extended number (e.g. 00000010ab314d19) being 'optimally encoded (e.g. 00000010ab314d19 -> 10ab314d19)

08 - Stack Data Queries

Stack data queries are opcodes that return information directly relating to the stack or to the data item at the top of the stack.

Stack Depth Check

The opcode OP_DEPTH will leave a value on the stack which indicates the depth of the stack. For example, if the stack had 3 items on it, calling OP_DEPTH would leave the integer '3' as a new item at the top of the stack.

Word	Input	Output	Description
OP_DEPTH	items	items, qty items	Counts the number of stack items onto the stack and places the value on the top

Insert Chapter 3 animation 9

Example:

```
OP_DEPTH OP_1 OP_EQUAL OP_IF
    <pubkey>
OP_ELSE
    OP_DUP OP_HASH160 <pubkeyhash> OP_EQUALVERIFY
OP_ENDIF
OP_CHECKSIG
```

In this example, the script checks the depth of the stack. If there is just 1 item on the stack, it assumes it is a signature and checks it against a public key stored in the script. If there is more than one item on the stack, it assumes this is a public key and signature, which it checks against a stored public key hash.

Size Query

OP_SIZE evaluates the first item on the stack and adds its length to the top of the stack.

Word	Input	Output	Description
OP_SIZE	item	item, item size	Pushes the string length of the top element of the stack (with popping it).

Insert Chapter 3 animation 10

Example:

```
OP_SIZE OP_10 OP_LESSTHANOREQUAL OP_VERIFY
```

In this example, the script checks that an item input to the stack is less than or equal to 10 bytes long. If the string is longer than 10 bytes long, the script fails. This can be useful when testing input conditions from user entered forms that are directly submitted to script.

Chapter 3 animation pack 9

Animate the following functions:

Word	Input	Output	Description
OP_DEPTH	Nothing	<Stack size>	Counts the number of stack items onto the stack and places the value on the top

Chapter 3 animation pack 10

Animate the following functions:

Word	Input	Output	Description
OP_SIZE	nothing	item size	Pushes the string length of the top element of the stack (with popping it).

Chapter 3 assessment 4

Assessment

1. Which script will be successful if `<input>` is `0x1234`
    1. `<input> OP_1 OP_SPLIT OP_4 OP_EQUALVERIFY OP_1 OP_SPLIT OP_3 OP_EQUALVERIFY OP_1 OP_SPLIT OP_2 OP_EQUALVERIFY`
    2. `<input> OP_1 OP_2 OP_3 OP_4 OP_CAT OP_CAT OP_CAT OP_EQUALVERIFY OP_TRUE`
    3. `<input> OP_SIZE OP_2 OP_EQUALVERIFY OP_1 OP_SPLIT OP_SWAP 0x12 OP_EQUALVERIFY 0x34 OP_EQUALVERIFY OP_TRUE` <-
    4. `<input> OP_1 OP_SPLIT OP_1 OP_SPLIT OP_1 OP_SPLIT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT 0x4321 OP_EQUALVERIFY OP_TRUE`
2. Which script will be unsuccessful if `<input1>`, `<input2>` and `<input3>` are `0x12`, `0x34` and `0x56` respectively?
    1. `0x56 OP_EQUALVERIFY 0x34 OP_EQUALVERIFY 0x12 OP_EQUALVERIFY OP_TRUE`
    2. `OP_ROT OP_CAT OP_CAT OP_SIZE OP_CAT OP_16 OP_NUM2BIN 0x345612 OP_EQUALVERIFY` <-
    3. `OP_2DROP`
    4. `OP_3DUP OP_3 OP_ROLL OP_EQUALVERIFY OP_2 OP_ROLL OP_EQUALVERIFY OP_EQUALVERIFY OP_TRUE`

09 - Bitwise transformations and Arithmetic

Bitcoin Script has a full array of bitwise/binary transformation and arithmetic opcodes.

Bitwise transformation opcodes

Bitwise transformation opcodes are used to perform bitwise transformations on data items on the stack.

Word	Input	Output	Description
OP_INVERT	in	out	Flips all of the bits in the input.
OP_AND	x1 x2	out	Boolean *and* between each bit in the inputs.
OP_OR	x1 x2	out	Boolean *or* between each bit in the inputs.
OP_XOR	x1 x2	out	Boolean *exclusive or* between each bit in the inputs.

Insert Chapter 3 animation 11

Checking bitwise outcomes

To check Bitwise outcomes, the scripting language provides an equality checker. For these checks to return true, the values on the stack must be equal at a bitwise level - i.e. they must be the same value and the same size.

Word	Input	Output	Description
OP_EQUAL	x1 x2	True / false	Returns 1 if the inputs are exactly equal, 0 otherwise.
OP_EQUALVERIFY	x1 x2	Nothing / *fail*	Same as OP_EQUAL, but runs OP_VERIFY afterward.

Insert Chapter 3 animation 12

Example 1:

```
<0x80> OP_AND <0x80> OP_EQUALVERIFY
```

This example checks that the first bit of the top stack item is 1 (`0x80 = 0b10000000`). If this condition

is not met, the script will fail. The other bits are not evaluated. The top stack item must be 1 byte long or the script will fail.

Example 2:

```
<0x7f> OP_OR <0x7f> OP_EQUALVERIFY
```

This example checks that the first bit of the top stack item is 0 (`0x7F = 0b01111111`). If this condition is not met, the script will fail. The other bits are not evaluated. The top stack item must be 1 byte long or the script will fail.

Example 3:

```
OP_XOR <0x7F> OP_OR <0xFF> OP_EQUALVERIFY
```

This example first applies an XOR function to two single byte values at the top of the stack. It then uses `0x7f OP_OR` to check that only one of them had the first bit set, and excludes the remaining bits from the test. If this condition is not met, the script will fail. The stack items must be 1 byte long or the script will fail.

Arithmetic transformations

These operations perform arithmetic and other mathematical transformations on data items on the stack.

Word	Input	Output	Description
OP_NEGATE	in	out	The sign of the input is flipped.
OP_ABS	in	out	The input is made positive.
OP_ADD	a b	out	a is added to b.
OP_1ADD	in	out	1 is added to the input.
OP_SUB	a b	out	b is subtracted from a.
OP_1SUB	in	out	1 is subtracted from the input.
OP_NOT	in	out	If the input is 0 or 1, it is flipped. Otherwise the output will b
OP_MUL	a b	out	a is multiplied by b.
OP_DIV	a b	out	a is divided by b.
OP_MOD	a b	out	Returns the remainder after dividing a by b.

Insert Chapter 3 animation 13

Example 1:

```
OP_DUP OP_DUP OP_MUL OP_MUL OP_ABS
```

In this example, the number at the top of the stack is duplicated twice and then the three values are multiplied together to calculate the cube of the original value. The absolute value of the outcome is the final result.

Example 2:

```
OP_DUP OP_MUL <314> OP_MUL <100> OP_DIV <1,000,000>* OP_LESSTHANOREQUAL
    OP_FALSE OP_RETURN
OP_ENDIF
```

In this example, a trusted IoT measurement device records and reports the radius of a circle. The script uses this value to calculate the circle's area using the equation Pi x r^2. If the circle is less than 1m^2 the script fails. Note that there are no floating point calculations in Bitcoin, so 314 is used to approximate Pi (3.14) and the resulting output divided by 100 to get a figure in m^2.

* For readability, <314>, <100> and <1,000,000> are shown as decimal numbers. To implement this example in script, the little endian integers 0x3A01, 0x64 and 0x40420F should be used respectively.

Example 3:

```
OP_DUP OP_SHA256 <1,000,000>* OP_MOD OP_NOTIF
    <1>
OP_ELSE
    <2>
OP_ENDIF
<pubkey1> <pubkey2> <2> OP_CHECKMULTISIG
```

* For readability <1,000,000> is shown as a decimal number. To implement this example in script, the little endian integer 0x40420F should be used.

In this example, the signing parties play a game where the first to find a signature that with a SHA256 hash that is divisible by 1 million can spend the output. To begin the game, the coin is spent into a non-final script using both keys which pays out equally. If one user finds a valid solution before the time expires, they can finalise the payment channel and spend the full amount to their own wallet.

Valid solutions to the script are as follows:

`<x> <signature 1>` (where signature 1 is divisible by 1,000,000)

`<x> <signature 2>` (where signature 2 is divisible by 1,000,000)

`<x> <signature 1> <signature 2>`

Boolean Shifts

Boolean shifts are used to shift the bits within a data item on the stack. There is no limit to the size of data item that these opcodes can be applied to.

Word	Input	Output	Description
OP_LSHIFT	a b	out	Logical left shift b bits. Sign data is discarded
OP_RSHIFT	a b	out	Logical right shift b bits. Sign data is discarded

Insert Chapter 3 animation 14

Boolean Data Checks

These opcodes check inputs against Boolean and/or conditions. In all cases, any non-zero value is considered a true/1, while a zero value of any length bytevector is considered a false/0.

Word	Input	Output	Description
OP_BOOLAND	a b	out	If both a and b are not 0, the output is 1. Otherwise 0.
OP_BOOLOR	a b	out	If a or b is not 0, the output is 1. Otherwis

Insert Chapter 3 animation 15

Arithmetic value checks

These opcodes allow the script to check stack values against pre-set/expected results to create complex functionality.

Word	Input	Output	Description

Word	Input	Output	Description
OP_NUMEQUAL	a b	out	Returns 1 if the numbers are equal, 0 otherwise.
OP_NUMEQUALVERIFY	a b	Nothing / *fail*	Same as OP_NUMEQUAL, but runs OP_VERIFY afterward.
OP_NUMNOTEQUAL	a b	out	Returns 1 if the numbers are not equal, otherwise.
OP_LESSTHAN	a b	out	Returns 1 if a is less than b, 0 otherwise
OP_GREATERTHAN	a b	out	Returns 1 if a is greater than b, 0 otherwise.
OP_LESSTHANOREQUAL	a b	out	Returns 1 if a is less than or equal to b, otherwise.
OP_GREATERTHANOREQUAL	a b	out	Returns 1 if a is greater than or equal to 0 otherwise.

Insert Chapter 3 animation 16

Max, Min and Within

These opcodes allow the script to select a maximum or minimum value from a group, or to discover whether a value is within a range.

Word	Input	Output	Description
OP_MIN	a b	out	Returns the smaller of a and b.
OP_MAX	a b	out	Returns the larger of a and b.
OP_WITHIN	x min max	out	Returns 1 if x is within the specified range (left-inclusive) otherwise.

Insert Chapter 3 animation 17

Chapter 3 animation pack 11

Animate the following functions:

Word	Input	Output	Description
OP_INVERT	in	out	Flips all of the bits in the input.
OP_AND	x1 x2	out	Boolean *and* between each bit in the inputs.
OP_OR	x1 x2	out	Boolean *or* between each bit in the inputs.
OP_XOR	x1 x2	out	Boolean *exclusive or* between each bit in the inputs.

Chapter 3 animation pack 12

Animate the following functions:

Word	Input	Output	Description
OP_EQUAL	x1 x2	True / false	Returns 1 if the inputs are exactly equal, 0 otherwise.
OP_EQUALVERIFY	x1 x2	Nothing / *fail*	Same as OP_EQUAL, but runs OP_VERIFY afterward.

Chapter 3 animation pack 13

Animate the following functions:

Word	Input	Output	Description
OP_NEGATE	in	out	The sign of the input is flipped.
OP_ABS	in	out	The input is made positive.
OP_ADD	a b	out	a is added to b.
OP_1ADD	in	out	1 is added to the input.
OP_SUB	a b	out	b is subtracted from a.
OP_1SUB	in	out	1 is subtracted from the input.
OP_NOT	in	out	If the input is 0 or 1, it is flipped. Otherwise the output will b
OP_MUL	a b	out	a is multiplied by b.
OP_DIV	a b	out	a is divided by b.
OP_MOD	a b	out	Returns the remainder after dividing a by b.

Chapter 3 animation pack 14

Animate the following functions:

Word	Input	Output	Description
OP_LSHIFT	a b	out	Logical left shift b bits. Sign data is discarded
OP_RSHIFT	a b	out	Logical right shift b bits. Sign data is discarded

Chapter 3 animation pack 15

Animate the following functions:

Word	Input	Output	Description
OP_BOOLAND	a b	out	If both a and b are not 0, the output is 1. Otherwise 0.
OP_BOOLOR	a b	out	If a or b is not 0, the output is 1. Otherwis

Chapter 3 animation pack 16

Animate the following functions:

Word	Input	Output	Description
OP_NUMEQUAL	a b	out	Returns 1 if the numbers are equal, 0 otherwise.
OP_NUMEQUALVERIFY	a b	Nothing / *fail*	Same as OP_NUMEQUAL, but runs OP_VERIFY afterward.
OP_NUMNOTEQUAL	a b	out	Returns 1 if the numbers are not equal, otherwise.
OP_LESSTHAN	a b	out	Returns 1 if a is less than b, 0 otherwise
OP_GREATERTHAN	a b	out	Returns 1 if a is greater than b, 0 otherwise.
OP_LESSTHANOREQUAL	a b	out	Returns 1 if a is less than or equal to b, otherwise.
OP_GREATERTHANOREQUAL	a b	out	Returns 1 if a is greater than or equal to 0 otherwise.

Chapter 3 animation pack 17

Animate the following functions:

Word	Input	Output	Description
OP_MIN	a b	out	Returns the smaller of a and b.
OP_MAX	a b	out	Returns the larger of a and b.
OP_WITHIN	x min max	out	Returns 1 if x is within the specified range (left-inclusive) otherwise.

Chapter 3 assessment 5

Assessment

1. Which set of inputs will **not** solve the following script? `OP_DEPTH OP_3 OP_EQUAL OP_IF OP_ADD OP_ADD OP_ELSE OP_DEPTH OP_2 OP_EQUAL OP_IF OP_MUL OP_ELSE OP_10 OP_DIV OP_ENDIF OP_ENDIF OP_16 OP_EQUAL`
    1. `0x04 0x08 0x04`
    2. `0x04 0x04`
    3. `0x12 0x02 0x02` <-
    4. `0xA0`
2. Which opcodes are needed to fill in the blanks for the following script to solve correctly when the input `0x0102030405060708` is used? `OP_3 OP_SPLIT OP_1 OP_SPLIT OP_SWAP OP_DUP OP_DUP OP_CAT OP_CAT ______ OP_ADD OP_3 OP_SPLIT OP_SWAP ______`
    1. `OP_OVER, OP_GREATERTHAN`
    2. `OP_ROT, OP_EQUALVERIFY` <-
    3. `OP_NIP, OP_EQUAL`
    4. `OP_MAX, OP_GREATERTHAN`

10 - Cryptographic Functions

Bitcoinscript includes a range of cryptographic functions such as hashing functions and ECDSA signature checks.

Hash Functions

There are three hash functions available in Bitcoin script:

1. RIPEMD160
2. SHA1
3. SHA256

There are also opcodes that perform double hash operations using these same base opcodes.

Each hash function consumes the topmost data item on the stack and replace it with the hash of that data item.

Word	Input	Output	Description
OP_RIPEMD160	in	hash	The input is hashed using RIPEMD-160.
OP_SHA1	in	hash	The input is hashed using SHA-1.
OP_SHA256	in	hash	The input is hashed using SHA-256.
OP_HASH160	in	hash	The input is hashed twice: first with SHA-256 and then with RIPEMD-160.
OP_HASH256	in	hash	The input is hashed two times with SHA-256.

ECDSA Signature Check Functions

Bitcoin script provides users with both single and multisignature options based on the Secp256k1 elliptic curve. These functions are core to the Peer to Peer functionality of Bitcoin, allowing users to assert ownership of coins when transacting.

It is important to understand that digital signatures alone cannot be used to establish identity, and there is an implicit requirement for users on each side of a transaction to take responsibility for checking/proving counterparty identity, and then using digital signatures to finalise the transactions.

The messages used to generate the signatures are created using reproducible data, which allows the nodes on the network to perform signature checks using only the transaction data itself. This is a key element of Bitcoin's peer-to-peer functionality, and allowing nodes to perform independent signature validations without user input.

Word	Input	Output	Description
OP_CHECKSIG	sig pubkey	True / false	The entire transaction's outpu inputs, and script (f the most recently-executed OP_CODESEPAR OR to the end) are hashed. The signat used by OP_CHECKSIG m be a valid signature this hash and publi key. If it is, 1 is returned, 0 otherwi
OP_CHECKSIGVERIFY	sig pubkey	Nothing / *fail*	Same as OP_CHECKSIG, b OP_VERIFY is executed afterward
OP_CHECKMULTISIG	x sig1 sig2 ... <n> pub1 pub2 ... <m>	True / False	Compares the first signature against e public key until it fir an ECDSA match. Starting with the subsequent public key, it compares the second signature against each remaining public ke until it finds an ECDSA match. The process is repeated until all signatures have been checked not enough public keys remain to produce a success result. All signature need to match a pu key. Because publi keys are not check again if they fail an signature comparis signatures must be placed in the script

			using the same ord as their correspond public keys were placed in the scriptPubKey or redeemScript. If all signatures are valic is returned, 0 otherwise. Due to a bug, an extra unuse value (x) is remove from the stack. Scri spenders must account for this by adding a junk value (typically zero) to th stack.
OP_CHECKMULTISIGV ERIFY	x sig1 sig2 ... <n> pub1 pub2 ... <m>	Nothing / *fail*	Same as OP_CHECKMULT G, but OP_VERIFY executed afterward

OP_CODESEPARATOR

`OP_CODESEPARATOR` is used in a Bitcoin script to indicate to the node checking the signature exactly which part of the scriptPubKey is being signed. When transactions are submitted to the network, the node inserts `OP_CODESEPARATOR` at the junction between input and output.

When the transaction validation engine reaches an ECDSA checking function, the message that it uses to perform the signature check only includes the script that comes after the most recent `OP_CODESEPARATOR` in the scriptPubKey being signed.

This functionality can be used in complex scripts to allow users to omit parts of the transaction output being signed, which can be useful when building complex functionality such as contracts with signature witness statements

Word	Input	Output	Description
OP_CODESEPARATOR	Nothing	Nothing	All of the signature checking words wil only match signatu to the data after the most recently-executed OP_CODESEPAR OR.

Example: Masked document for witness signature

scriptSig: `<witness_signature> <witness_public_key> <witness_name> <signatory_signature>`

scriptPubKey: `OP_CODESEPARATOR <hash_contract> OP_DROP <signatory_public_key> OP_CHECKSIGVERIFY OP_CODESEPARATOR <witness_statement> OP_2DROP OP_CHECKSIG`

In this example, an output is created containing a contract document that is ready to be signed into effect as well as a witness statement allowing a third party to attest that they witnessed the signing party performing the contract signature.

The scriptSig contains 4 elements. First, the witness signature, public key and name are added, then the signing party's signature.

The first `OP_CODESEPARATOR` is inserted by the validation engine when the transaction scriptSig and scriptPubKey are joined. The scriptSig contains both the witness signature and signatory signature. During the validation processs, a hash of the document being signed (`<hash_contract>`) is pushed onto the stack and dropped before the signatory's public key is pushed onto the stack and their signature checked using `OP_CHECKSIGVERIFY`. When OP_CHECKSIGVERIFY is processed, the message signed by the signature must include the scriptPubKey all the way back to the first `OP_CODESEPARATOR` which includes the contract, their public key and the witness statement.

After the first OP_CHECKSIGVERIFY, `OP_CODESEPARATOR` is used.

The script then pushes a witness statement onto the stack, drops it and the witness' name, and then checks their signature against the public key they provided. Again, the message in the signature includes the script as far back as the most recent OP_CODESEPARATOR. This means that the message signed by the witness does not include the contract itself, allowing the witness to act as a witness of the signing party's identity and signature only. This can be likened to showing the witness the signing page of a contract and letting them watch you sign, but keeping the remainder private.

Chapter 3 video 6

Cryptographic features

Script	Video
Bitcoin's cryptographic functions exist as a set of tools within the protocol that allow peers who manage their identity off-chain to show their intent via the blockchain.	Todd
The functions are broken into two groups: Hash functions, and signature checks using the Elliptic Curve Digital Signature Algorithm.	Animate table 1 below script
It is important to note that there are no encryption features in Bitcoin, and 100% of the Bitcoin protocol is in plain text.	Todd Show 'no encryption' in text
Although the Bitcoin protocol itself does not include encryption features, it does allow for the insertion of data into transactions, including the possibility of incorporating encrypted data.	Todd
However, for the encrypted data inserted into Bitcoin transactions to be utilized, a second-layer system is required. This system enables the data owner to locate and decrypt the information at a later date.	Show encrypted data being pushed into a transaction script and then used by a second orc system

Table 1:

Hash Functions	ECDSA Checks
OP_RIPEMD160	OP_CHECKSIG
OP_SHA1	OP_CHECKSIGVERIFY
OP_SHA256	OP_CHECKMULTISIG
OP_HASH256 (Double SHA256)	OP_CHECKMULTISIGVERIFY
OP_HASH160 (SHA256 + RIPEMD160)	

Chapter 3 animation pack 18

Animate the following functions:

Word	Input	Output	Description
OP_RIPEMD160	in	hash	The input is hashed using RIPEMD-160.
OP_SHA1	in	hash	The input is hashed using SHA-1.
OP_SHA256	in	hash	The input is hashed using SHA-256.
OP_HASH160	in	hash	The input is hashed twice: first with SHA-256 and then with RIPEMD-160.
OP_HASH256	in	hash	The input is hashed two times with SHA-256.

Chapter 3 animation pack 19

Animate the following functions:

Word	Input	Output	Description
OP_CHECKSIG	sig pubkey	True / false	Depending on the SIGHASH flag, some or all of the transaction's outputs, inputs, and script (from th most recently-executed OP_CODESEPARATOR to the er are hashed. The signature used b OP_CHECKSIG must be a valid signature for this hash and public key. If it is, 1 is returned, 0 otherwi
OP_CHECKSIGVERIFY	sig pubkey	nothing/fail	Same as OP_CHECKSIG, but OP_VERIFY is executed afterwar
OP_CHECKMULTISIG	x sig1 sig2 … <no. signatures> pub1 pub2 <no. public keys>	True / False	Compares the first signature agair each public key until it finds an ECDSA match. Starting with the subsequent public key, it compare the second signature against each remaining public key until it finds a ECDSA match. The process is repeated until all signatures have been checked or not enough publ keys remain to produce a success result. All signatures need to matc public key. Because public keys a not checked again if they fail any signature comparison, signatures must be placed in the scriptSig us the same order as their corresponding public keys were placed in the scriptPubKey or redeemScript. If all signatures are valid, 1 is returned, 0 otherwise. D to a bug, an extra unused value (x removed from the stack. Script spenders must account for this by adding a junk value (typically zerc the stack.
OP_CHECKMULTISIGV ERIFY	x sig1 sig2 … <no. signatures> pub1 pub2 <no. public keys>	nothing/fail	Same as OP_CHECKMULTISIG, OP_VERIFY is executed afterwar

Chapter 3 animation 20

Animate the following function:

Word	Input	Output	Description
OP_CODESEPARATOR	Nothing	Nothing	All of the signature checking words will c match signatures to the data after the mo recently-executed OP_CODESEPARATOR.

11 - Disabled and Removed Opcodes

Due to changes made to the scripting language between 2010 and 2018, there are a small number of opcodes that have been disabled or permanently removed from the Bitcoin scriping language.

Removed opcodes are those who's hexadecimal instruction code has been re-used for opcodes with different functionality. The functionality expressed by all removed opcodes can be replicated with combinations of both new and existing opcodes, albeit with some cost in terms of the size of the script in certain cases. Since these opcodes no longer appear in the opcode set, we will not look any further at these.

Disabled opcodes retain their hexadecimal codes and are listed in the opcode set, however transactions that try to create outputs that use them are considered invalid and will be rejected by the network. There is no schedule for their re-introduction into the Bitcoin protocol, and the disablement may be permanent however the functionality they enabled is can be generated using combinations of other available opcodes.

OP_2MUL, OP_2DIV

OP_2MUL and OP_2DIV provide shortcuts for performing arithmetic functions on the stack.

To replicate the functionality of OP_2MUL, the following snippet can be used:

```
OP_2 OP_MUL
```

To replicate the functionality of OP_2DIV, the following snippet can be used:

```
OP_2 OP_DIV
```

OP_VER, OP_VERIF, OP_VERNOTIF

OP_VER's original functionality pushes the version number of the transaction onto the top of the stack. The technique required to replicate the function of this opcode is complicated, and will be covered in Chapter 5.

OP_VERIF's original functionality compares the version number of the transaction to the value on top of the stack, and causes the script to enter an IF loop if the transaction version is the same as the value on top of the stack. The technique required to replicate the function of this opcode is complicated, and will be covered in Chapter 5.

OP_VERNOTIF's original functionality compares the version number of the transaction to the value on top of the stack, and causes the script to enter an IF loop if the transaction version is different to the value on top of the stack. The technique required to replicate the function of this opcode is complicated, and will be covered in Chapter 5.

Chapter 3 assessment 6

Assessment

1. Bob and Alice and Carol create an account where either Alice or Carol can spend funds on their own, but Bob must have a countersignature from either Alice or Carol. Which script achieves this functionality? The solution generated by the wallet when Bob is spending money is: `OP_1 <alice/carol_sig> <bob_sig>`
    1. `OP_DEPTH OP_2 OP_EQUAL OP_NOTIF OP_1 <alice_pubkey> <carol_pubkey> OP_2 OP_CHECKMULTISIG OP_ELSE OP_3 <alice_pubkey> <carol_pubkey> <bob_pubkey> OP_3 OP_CHECKMULTISIG OP_ENDIF`
    2. `OP_DUP <bob_pubkey> OP_CHECKSIG OP_IF OP_DROP OP_ENDIF OP_1 <alice_pubkey> <carol_pubkey> OP_2 OP_CHECKMULTISIG` <-
    3. `OP_DEPTH OP_1 OP_EQUAL OP_DUP <alice_pubkey> OP_CHECKSIG OP_SWAP <carol_pubkey> OP_CHECKSIG OP_BOOLOR OP_ELSE OP_2 <bob_pubkey> <alice_pubkey> <carol_pubkey> OP_3 OP_CHECKMULTISIG OP_ENDIF`
    4. `OP_1 <alice_pubkey> <carol_pubkey> <bob_pubkey> OP_3 OP_CHECKMULTISIG`
2. Dan and Evelyn want a wallet signed by a 2of2 multisig that incorporates public key hashing so that their keys don't need to be written into the output when it is created. The script must check that the solution they provide is a valid 2of2 multisig evaluation. Use of both separate and combined hashes to check the pre-defined public keys are valid approaches. The required input solution is: `OP_1 <dan_signature> <evelyn_signature> OP_2 <dan_pk> <evelyn_pk>` Which script does NOT correctly achieve this functionality?
    1. `OP_OVER OP_HASH160 <dan_pkhash> OP_EQUALVERIFY OP_DUP OP_HASH160 <evelyn_pkh> OP_EQUALVERIFY OP_CHECKMULTISIG`
    2. `OP_3DUP OP_CAT OP_CAT OP_HASH160 <combined_hash> OP_EQUALVERIFY OP_2 OP_CHECKMULTISIG`
    3. `OP_DUP OP_HASH160 <evelyn_pkh> OP_EQUALVERIFY OP_OVER OP_HASH160 <dan_pkh> OP_EQUALVERIFY OP_2 OP_CHECKMULTISIG` <-
    4. `OP_DUP OP_TOALTSTACK OP_HASH160 <evelyn_pkh> OP_EQUALVERIFY OP_DUP OP_TOALTSTACK OP_HASH160 <dan_pkh> OP_DUP OP_2 OP_EQUALVERIFY OP_FROMALTSTACK OP_FROMALTSTACK OP_2 OP_CHECKMULTISIG`
3. From the perspective of the protocol, (e.g. ignoring usage rules/node policies) what is the maximum number of signatures that can be checked using OP_CHECKMULTISIG?
    1. 16
    2. No limit <-
    3. 65,535
    4. 4.3 billion
4. When a 3of5 signature check is performed using OP_CHECKMULTISIG, how many data items are consumed from the stack?
    1. 8
    2. 9
    3. 10

4. 11 <-

Chapter 4: SIMPLE SCRIPTS

01 - Introduction

In this chapter we will look at some simple script templates. Some of these are commonly used (e.g. P2PKH) and for those of you with some knowledge of Bitcoin will be familiar.

For each template, we will do a full breakdown of the script being solved in the evaluation engine including a table showing the state of the stack at each step, the opcodes being used and then present an animation to help you visualise the activity taking place.

Insert Chapter 4 video 1

Bitcoin script templates are an important part of your toolset and being able to create simple, efficient templates for the applications you are designing is a critical part of being an effective Bitcoin Script engineer.

Chapter 4 video 1

Script	Video
In this chapter we will take a detailed look at some simple script templates which can be used for wallets that require different account types.	Todd
These include:	Blank screen for text. Show at top of screen: Templates:
Pay to Public key scripts such as those commonly used in the very early days of the network	Add text: 1. Pay to Public Key (P2PK)
Pay to Hash Puzzle scripts, useful where simple, low overhead locking and unlocking is needed	Add text: 2. Pay to Hash Puzzle (P2HP)
Pay to Public Key Hash, the most widely used script template today	Add text: 3. Pay to Public Key Hash (P2PKH)
Pay to Multisignature, useful when creating accounts where multiple parties have access to tokens or funds	Add text: 4. Pay to Multisignature (P2MS)
Pay to Multisignature Hash, a version of a multisignature script that masks public keys until a spend event occurs	Add text: 5. Pay to Multisignature Hash (P2MSH)
And finally, a look at R-Puzzles, a technique that leverages the R-value in the Elliptic Curve Digital Signature Algorithm to provide an additional checking mechanism allowing two separate keys to be used in the same signature.	Add text: 6. Pay to Public Key (P2RP)
These techniques on their own provide for a multitude of use cases, however the engineer who understands these functions deeply will be able to blend and meld them into powerfully useful transaction types for a variety of applications.	Todd

01 - Pay to Public Key (P2PK)

Pay to Public Key is the most simple script that can be deployed that uses the security of Elliptic curve signatures to lock transaction outputs.

A pay to public key script is defined as follows:

`<public_key> OP_CHECKSIG`

To spend an output that is locked with a P2PK script, the following solution is provided:

`<signature>`

The validation engine will evaluate the full script as follows:

`<signature> <public_key> OP_CHECKSIG`

For the purposes of brevity, we exclude the pushdata opcodes and use | | to represent the OP_CODESEPARATOR that the validation engine automatically inserts inbetween the input and output scripts.

A breakdown of the script evaluation process is shown below:

Stack	Script	Description
Empty.	<sig> | | <pubKey> OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig>	<pubKey> OP_CHECKSIG	Signature is added to the stack.
<sig> <pubKey>	OP_CHECKSIG	Pubkey is added to stack.
true	Empty.	Signature is checked against the public key

Insert Chapter 4 Animation 1

As shown above, the signature and public key are pushed onto the stack, and then consumed by OP_CHECKSIG which leaves the result of the check on the stack. If the signature is valid, the input can be spent in the transaction.

Chapter 4 animation 1

See google doc

02 - Pay to Hash Puzzle

A Pay to Hash Puzzle script is a simple script that uses a hash function as the checking function, requiring the spending party to provide a valid solution to the puzzle to spend the output.

When processing hash puzzle scripts, the validation engine takes one or more input argument and process them against one or more hash/algebraic functions to create a script that checks that the spending party has knowledge of a piece of hidden information. These scripts have very low overhead, but reduced security. Care should be taken when creating outputs with these functions as some hash function opcodes (e.g. OP_SHA1) use hashing functions that are computationally insecure.

A pay to hash script can be defined in many ways:

Example 1: `OP_SHA256 <sha256_hash> OP_EQUAL`

Example 2: `OP_HASH256 <double_sha256_hash> OP_EQUAL`

Example 3`: OP_SHA256 OP_RIPEMD160 <ripemd160_hash> OP_EQUAL`

To spend an output that is locked with the script in example 3, the following solution is provided:

`<pre-hash_value>`

The validation engine will evaluate the full script of Example 3 as follows:

`<pre-hash_value> OP_SHA256 OP_RIPEMD160 <ripemd160_hash> OP_EQUAL`

A breakdown of the script evaluation process is shown below:

Stack	Script	Description
Empty.	<pre-hash_value> | | OP_SHA256 OP_RIPEMD160 <ripemd160_hash> OP_EQUAL	scriptSig and scriptPubKey are combined.
<pre-hash_value>	OP_SHA256 OP_RIPEMD160 <ripemd160_hash> OP_EQUAL	Pre-hash value is added to the sta
<sha256_hash>	OP_RIPEMD160 <ripemd160_hash> OP_EQUAL	Pre-hash value is hashed using SHA256 hashing algorithm
<ripemd160_hash>	<ripemd160_hash> OP_EQUAL	SHA256 hash of pre-hash value is hashed with RIPEMD160 hash algorithm
<ripemd160_hash> <ripemd160_hash>	OP_EQUAL	Expected RIPEMD160 hash is ad to the stack
true	Empty.	Hash value is checked against the expected hash value

Insert Chapter 4 animation 2

As shown above, the pre-hash value is pushed onto the stack, before being double hashed, first with the SHA256 hashing algorithm and subsequently with the RIPEMD160 hashing algorithm. If the correct pre-hash value is used, the double hash value will match the expected hash outcome, and the input can be spent in the transaction.

Pay to Script Hash (P2SH)

Pay to Script Hash (P2SH) was a so-called 'soft fork' change introduced to Bitcoin in 2012 ostensibly to allow more complex scripts to be used with smaller transactions. The scheme uses protocol hacks to bypass the expected behaviour of the network, and was disallowed following the 2020 Genesis upgrade on the BitcoinSV network.

Invalid P2SH outputs are defined using the following template:

```
OP_HASH160 <script_hash> OP_EQUAL
```

This script performs the same process outlined in Example 3 above, but uses the OP_HASH160 opcode to perform the double hash with a single instruction.

Because P2SH introduces functionality that is not compatible with the Bitcoin protocol, nodes on the BSV network will evaluate any transaction sent to the network that creates a P2SH output as invalid, and will reject it with the 'Error: Pay to Script Hash' error code.

Chapter 4 animation 2

Chapter 4 assessment 1

Have a crack son...

03 - Pay to Public Key Hash (P2PKH)

Pay to Public Key Hash (P2PKH) is the most widely used locking script on the blockchain today. It combines the security of elliptic curve signatures with a hash function. The major benefit over the use of a P2PK script is that the user's public key is kept private from the network until the output is spent, providing an additional layer of cryptographic security for the user. The major benefit over the use of a Pay to Hash output is the use of Elliptic curve signatures, which are more secure than hash functions alone.

A pay to public key hash script is defined as follows:

```
OP_DUP OP_HASH160 <public_key_hash> OP_EQUALVERIFY OP_CHECKSIG
```

To spend an output that is locked with a P2PKH script, the following solution is provided:

```
<signature> <public_key>
```

The validation engine will evaluate the full script as follows:

```
<signature> <public_key> OP_DUP OP_HASH160 <public_key_hash>
OP_EQUALVERIFY OP_CHECKSIG
```

A breakdown of the script evaluation process is shown below:

Stack	Script	Description
Empty.	<sig> <pubKey> | | OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig> <pubKey>	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Signature and public key are adde to the stack
<sig> <pubKey> <pubKey>	OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Public key is duplicated.
<sig> <pubKey> <pubKeyHash>	<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Duplication of public key is hashe
<sig> <pubKey> <pubKeyHash> <pubKeyHash>	OP_EQUALVERIFY OP_CHECKSIG	Expected public key hash is adde the stack
<sig> <pubKey>	OP_CHECKSIG	Equality is checked between the script generated public key hash a expected public key hash.
true	Empty.	The signature is checked against public key

Insert Chapter 4 animation 3

As shown above, the signature and public key are both provided by the spending party. The public key is hashed, and the hash is checked against an expected value stored in the output. The check uses OP_EQUALVERIFY, automatically failing the script if the check is not equal. Once it is established that the public key hashes to the expected value, OP_CHECKSIG checks the signature and leaves the result of the check on the stack. If the signature is valid, the input can be spent in the transaction.

Chapter 4 animation 3

04 - Pay to MultiSig (P2MS)

Pay to MultiSig (P2MS) uses the OP_CHECKMULTISIG opcode to perform a check against multiple signatures and public keys. There are no limits to the number of keys/signatures that can be checked beyond any limits imposed by nodes on the size or complexity of individual transactions.

For this example, we will use a 2of3 multisignature operation. A 2of3 multisig script is defined as follows:

`2 <pubkey_1> <pubkey_2> <pubkey_3> 3 OP_CHECKMULTISIG`

To spend an output that is locked with a 2of3 P2MS script, any of the following solutions may be provided:

Example 1: `1 <signature_1> <signature_2>`

Example 2: `1 <signature_1> <signature_3>`

Example 3: `1 <signature_2> <signature_3>`

Using the example 1, the validation engine will evaluate the full script as follows:

`1 <signature_1> <signature_2> 2 <pubkey_1> <pubkey_2> <pubkey_3> 3 OP_CHECKMULTISIG`

A breakdown of the script evaluation process is shown below:

Stack	Script	Description
Empty.	1 <signature_1> <signature_2> | | 2 <pubkey_1> <pubkey_2> <pubkey_3> 3 OP_CHECKMULTISIG	scriptSig and scriptPubKey are combined.
1 <signature_1> <signature_2>	2 <pubkey_1> <pubkey_2> <pubkey_3> 3 OP_CHECKMULTISIG	Signatures are added to the stack
1 <signature_1> <signature_2> 2 <pubkey_1> <pubkey_2> <pubkey_3> 3	OP_CHECKMULTISIG	Public keys and multi signature evaluation criteria are added to th stack
true	Empty.	Multi signature evaluation is performed

Insert Chapter 4 animation 4

As shown above, the spending parties must supply 2 valid signatures signed with keys taken from the pool of three keys stored in the output being spent. The signatures must be provided in the same order corresponding to the public keys in the locking script.

e.g. if the spending party submitted `1 <signature_2> <signature_1>` as their solution, this would be invalid and the transaction would be rejected.

Junk Bug

The eagle eyed among you would have noticed that each of the example solutions has a single '1' as the first item in the script. This is a result of a bug in the evaluation software for OP_CHECKMULTISIG in the original node client. The bug is harmless, and results in minimal overhead to users. To maintain the protocol, the bug has been left in all subsequent versions of the node client and will remain as part of the Bitcoin protocol. This stack item is not checked in the evaluation process and as such can be any value of any valid length. Transactions submitted to the network that do not have an extra stack item at the top of the stack are considered invalid and will be rejected.

Chapter 4 animation 4

Chapter 4 assessment 2

Have another crack

05 - Pay to MultiSignature Hash (P2MSH)

Stack	Script	Description
Empty.	1 <signature_1> <signature_2> <pubkey_1> <pubkey_2> <pubkey_3> | |OP_3DUP OP_CAT OP_CAT OP_HASH160 <3pubkey_hash> OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_2 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	scriptSig and scriptPubKey a combined.
1 <signature_1> <signature_2> <pubkey_1> <pubkey_2> <pubkey_3>	OP_3DUP OP_CAT OP_CAT OP_HASH160 <3pubkey_hash> OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_2 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	Signatures and public keys a added to the stack
1 <signature_1> <signature_2> <pubkey_1> <pubkey_2> <pubkey_3><pubkey_1> <pubkey_2> <pubkey_3>	OP_CAT OP_CAT OP_HASH160 <3pubkey_hash> OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_2 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	3 public keys are duplicated
	OP_CAT OP_HASH160 <3pubkey_hash>	

1 <signature_1> <signature_2> <pubkey_1> <pubkey_2> <pubkey_3><pubkey_1> <pubkey_2_3_cat>	OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_2 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	Pubkey 2 and Pubkey 3 are concatenated
1 <signature_1> <signature_2> <pubkey_1> <pubkey_2> <pubkey_3> <pubkey_1_2_3_cat>	OP_HASH160 <3pubkey_hash> OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_2 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	Pubkey 1 is concatenated wi Pukey 2_3 concatenation
1 <signature_1> <signature_2> <pubkey_1> <pubkey_2> <pubkey_3> <pubkey_1_2_3_hash>	<3pubkey_hash> OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_2 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	Concatenated pubkeys are double hashed using OP_HASH160
1 <signature_1> <signature_2> <pubkey_1> <pubkey_2> <pubkey_3> <pubkey_1_2_3_hash> <3pubkey_hash>	OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_2 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	Expected 3 pubkey hash is pushed onto the stack
1 <signature_1> <signature_2> <pubkey_1> <pubkey_2> <pubkey_3>	OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_2 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	Generated hash is evaluated against expected hash
1 <signature_1> <signature_2> Altstack:<pubkey_3> <pubkey_2> <pubkey_1>	OP_2 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	Pubkeys 3, 2 and 1 are push to altstack

1 <signature_1> <signature_2> 2Altstack:<pubkey_3> <pubkey_2> <pubkey_1>	OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_CHECKMULTISIG	Constant value '2' is pushed onto the stack
1 <signature_1> <signature_2> 2 <pubkey_1><pubkey_2> <pubkey_3>	OP_3 OP_CHECKMULTISIG	Pubkeys 3, 2 and 1 are pulle from altstack
1 <signature_1> <signature_2> 2 <pubkey_1><pubkey_2> <pubkey_3> 3	OP_CHECKMULTISIG	Constant value '3' is pushed onto the stack
true	Empty.	Multisignature evaluation is performed

Insert Chapter 4 animation 5

As shown above, the spending parties must supply 2 valid signatures and the three public keys that correspond to the hash stored in the output. The 3 public keys are duplicated and hashed using the OP_HASH160 opcode, and the resultant data item is checked against a hash value stored in the output.

Following this, the three public keys are pushed to the ALTSTACK so that the 2 signature evaluation criteria can be inserted, and then pulled back to the main stack. The 3 pubkey evaluation criteria is then added and the multisig operation takes place.

These types of scripts can be applied in many ways to enable multi-party collaboration for a range of use cases.

Chapter 4 animation 5

06 - R-Puzzles

R-Puzzles are a means by which the ephemeral key used in the calculation of an elliptic curve signature can be applied to a locking function. R-Puzzles provide additional capabilities to users of the Bitcoin protocol by allowing parties to provide the ephemeral keys to third parties to access specific outputs on the blockchain so that valid signatures can be created.

To achieve this, the R-Puzzle script breaks down a signature in Bitcoin's modified DER format and extracts the R-value, before evaluating it against a value stored in the output script. The Bitcoin DER format is as follows:

Data Item	Example data / notes
Sequence Identifier	0x30
Length of Sequence	0x44 (Variable length, max value 0x46)
Integer Identifier	0x02
Byte-length of r	0x20 (Variable length, max value 0x21)
r	e9d34347e597e8b335745c6f8353580f4cbdb4bcde2794ef7aab915d9966 (Must not be negative)
Integer identifier	0x02
Byte-length of s	0x20 (Variable length, max value 0x21)
s	4f2ccb52c7243c55bde34934bd55efbdac21c74a20bb7b438d1b6de3311f (Low signature coordinate only)
Sighash type	0x41 (SIGHASH_ALL | SIGHASH_FORKID)

Insert Chapter 4 animation 6

An R-Puzzle script is defined as follows:

`OP_3 OP_SPLIT OP_NIP OP_1 OP_SPLIT OP_SWAP OP_SPLIT OP_DROP <r-value> OP_EQUAL`

Insert Chapter 4 animation 7

Pay to R-Puzzle Hash

This script does not include a signature evaluation so a further extension combines a signature check and hash function into a script called Pay to R-Puzzle Hash (P2RPH).

A P2RPH script is defined as follows:

```
OP_OVER OP_3 OP_SPLIT OP_NIP OP_1 OP_SPLIT OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG
```

To spend an output that is locked with a P2RPH script, the following solution is provided:

```
<signature> <public_key>
```

The validation engine will evaluate the full script as follows:

```
<signature> <public_key> OP_OVER OP_3 OP_SPLIT OP_NIP OP_1 OP_SPLIT OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG
```

A breakdown of the script evaluation process is shown below:

Stack	Script	Description
Empty.	<sig> <pubKey> | | OP_OVER OP_3 OP_SPLIT OP_NIP OP_1 OP_SPLIT OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig> <pubKey>	OP_OVER OP_3 OP_SPLIT OP_NIP OP_1 OP_SPLIT OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	Signature and public key are adde to the stack
<sig> <pubKey> <sig>	OP_3 OP_SPLIT OP_NIP OP_1 OP_SPLIT OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	Signature is copied to top of stack
<sig> <pubKey> <sig> 3	OP_SPLIT OP_NIP OP_1 OP_SPLIT OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	Constant value '3' is added to top stack.
<sig> <pubKey> <sig_l3> <sig_r>	OP_NIP OP_1 OP_SPLIT OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	First 3 bytes of DER encoded signature are split from remainder
<sig> <pubKey> <sig_r>	OP_1 OP_SPLIT OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	First 3 bytes of DER encoded signature are removed from stack
<sig> <pubKey> <sig_r> 1	OP_SPLIT OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	Constant value '1' is added to stac
<sig> <pubKey> <r_len> <sig_r>	OP_SWAP OP_SPLIT OP_DROP OP_HASH160 <r_hash>	The length of r is split from the signature remainder

	OP_EQUALVERIFY OP_SPLIT OP_DROP OP_CHECKSIG	
<sig> <pubKey> <sig_r> <r_len>	OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	Length of r is moved to top of stack
<sig> <pubKey> <r> <sig_r>	OP_DROP OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	r value is split from signature remainder
<sig> <pubKey> <r>	OP_HASH160 <r_hash> OP_EQUALVERIFY OP_CHECKSIG	Signature remainder is dropped fr stack
<sig> <pubKey> <hash160_r>	<r_hash> OP_EQUALVERIFY OP_CHECKSIG	r value is hashed using OP_HASH160
<sig> <pubKey> <hash160_r> <r_hash>	OP_EQUALVERIFY OP_CHECKSIG	Expected r-hash is added to the st
<sig> <pubKey>	OP_CHECKSIG	Expected r-hash is tested for equa against generated r-hash
true	Empty.	The signature is checked against public key

As shown above, the spending party must supply a valid signature generated from the public key also provided. Because we don't check the public key, this signature can be generated with any valid public key. The script copies the signature to the top of the stack before extracting the length of the r component, and using this length value to extract r itself. Once r has been separated from the signature, it is hashed and checked against an expected value before the signature is checked for validity against the public key provided. A wallet using R-Puzzles must keep a list of k-values which it needs to sign R-Puzzle inputs. These are mathematically identical to ECDSA public keys.

R-Puzzles can form a useful component of other second layer functionality and can be incorporated into more elaborate scripts for added functionality.

Chapter 4 animation 6

Chapter 4 animation 7

Chapter 4 assessment 3

Moar crack

Chapter 5: OP_PUSH_TX

01 - Turing Machines

Despite much dispute over the years, there are now several known methods of using Bitcoin script to create Turing complete programs and scripts. This chapter looks at the OP_PUSH_TX method.

A Turing machine is defined as a machine that manipulates symbols on a tape according to a set of rules. Despite the simplicity of this definition, Turing machines can be complex, and have been shown capable of performing any mathematical or computing function.

The tape used by a Turing machine is unbounded in size and divided into discrete cells, each of which holds one symbol from a finite set. The machine has a 'head' which is positioned over these cells, the contents of which defines the current state, and is used to calculate the next state. The machine moves in both directions on the tape, modifying each cell depending on the current state and the cell's contents. Once the calculation is complete, the process halts, ending the process.

In Bitcoin, we treat the ledger as the unbounded tape and use Bitcoin transactions as the cells. Each output that is part of the Turing machine contains script that reflects the current state and the evaluation process that determines the next state based on the transaction inputs. In this way, complex, multi-step operations can be developed which are programmed to move to one of a finite set of valid next states. This type of operation can also be called a 'Finite State Machine'.

Insert Chapter 5 video 1

There are several explanations and examples of Turing completeness in Bitcoin script, including the following:

Infinite and Unbounded
Craig Wright

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3160279

A Proof of Turing completeness in Bitcoin Script
Craig Wright

Turing Machine on Bitcoin
Medium

Each of these examples leverages a technique called `OP_PUSH_TX` which uses the properties of Bitcoin's digital signatures to discover the current state of a process, evaluate inputs and determine the next state. An agent is always required, and features such as payment channels can be leveraged to deliver demand based services within simple contracts.

In this chapter we will look at how this technique works, and evaluate the components of the Elliptic Curve signatures that enable these powerful applications to be developed.

Chapter 5 video 1

Script	Video
The history of the Turing machine predates	Todd
Alan Turing,	Show Alan Turing
and can be traced back to the origins of digital computing.	Todd with Alan Turing
The concept was initially theorised by Charles Babbage	Show Charles Babbage
who is recognised as the pioneer of digital computing.	Todd with Charles Babbage
Babbage's first implementation of a digital computer was the groundbreaking invention known as the Difference Engine. This remarkable device was capable of performing polynomial mathematics in the 1830s.	Show babbage Difference Engine
Turing machines are defined as an abstract machine that manipulates	Todd
symbols on a strip of tape according to a set of rules.	Show Turing strip animation e.g. https://www.youtube.com/watch?v=gJQTFhkhw but faster
In the context of Bitcoin, we can leverage the blockchain as the tape, creating transactions that evaluate information from a previous state to generate new outputs, effectively forming a state machine.	Show the tape morphing into blockchain tape w transactions
In this chapter, we will explore a key technique called OP_PUSH_TX, which allows for the construction of complex functionality using Bitcoin script as a Turing-complete language. This technique empowers developers to push transactions onto the blockchain, enabling dynamic evaluation and the execution of advanced operations within the Bitcoin ecosystem.	Todd - At the end show OP_PUSH_TX on scree

02 - Elliptic Curve Signatures in Bitcoin

Bitcoin signatures use a well known and widely used cryptographic technique known as Elliptic Curve Digital Signature algorithm (ECDSA). ECDSA was developed over many years and was accepted in 1998 as an ISO standard, in 1999 as an ANSI standard, and in 2000 as IEEE and NIST standards. ECDSA is closely linked to Elliptic Curve Encryption (ECE) and key pairs that are used to encrypt data with ECE can also be used to generate digital signatures with ECDSA.

Insert Chapter 5 video 2

An elliptic curve signature is generated by applying an elliptic curve function to a message. For the purposes of this exercise there is no direct need to understand the mathematics. Students who wish to gain further expertise in this area are urged to take the Bitcoin Primitives course, Introduction to Digital Signatures in Bitcoin.

For Bitcoin to work, the nodes that validate transactions must have a way of reconstructing the message signed by the user without the user having to provide anything but the Bitcoin transaction they are sending to the network.

This is achieved by making the message from a rigidly defined transaction pre-image that is comprised of a set of information that the node can extract directly from the transaction in order to reconstruct the message. That information is made up of the following set of 11 parameters:

Item	Name	Description, Length and format
1	nVersion	The version of the protocol being used to define this transaction - 4 Byte little-endian integer
2	hashPrevouts	either a SHA256 hash of the transaction's concatenated input source / sources or a 32 byte null string if SIGHASH_ANYONECANPAY is used 32 Byte hash string
3	hashSequence	the concatenation of the input nSequence value / values or a null string SIGHASH_ANYONECANPAY is used - 32 Byte hash string
4	outpoint	Outpoint for input being signed - concatenation of input TXID + Vout - 32 Byte transaction hash + 4 Byte little-endian integer
5	scriptLen	Locking script length - a VarInt - 1, 3, 5 or 9 bytes depending on script length
6	scriptPubKey	Locking script for output being spent - scriptLen byte string
7	value	Value of input - An integer representing the input's satoshi value - 8 Byte little-endian integer - 4 byte little-endian integer
8	nSequence	Then nSequence value for input being signed - 4 byte little-endian integ
9	hashOutputs	If SIGHASH_ALL is used, this value is the SHA256 hash of the concatenation of all transaction outputs in Transaction output format. If SIGHASH_SINGLE is used, this value is the SHA256 hash of only the output corresponding to the input's Vin index. If SIGHASH_NONE is use this is a null string - 32 Byte hash string
10	nLocktime	The nLocktime value for the transaction - 4 byte little-endian integer
11	sighash	The SIGHASH flags being applied to this signature - 4 bytes, XX000000 where XX is Sighash Flag

Show Chapter 5 animation 1

To calculate a signature the following mathematical data points are used:

1. A keypair In the sect256k1 calculated with:
   P = S · G
   where:
   P is the public key, calculated using:
   S, the Private key, an integer value in the range 0 - n where n =
   FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
   G, the Generator Point, a point on the elliptic curve with the following co-ordinate:
   x - 79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
   y - 483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8

2. Message m where:
   m is the transaction pre-image as defined on page 2

Using these datapoints the following method is applied:

1. Calculate R = k · G
   where k is typically a randomly generated number (called an ephemeral key) generated on demand. When using R-Puzzles, k is pre-known and is used as part of the evaluation process
2. Define r = x-coordinate of R
3. Calculate s = k^-1(H(m) + S * r)mod n
   Unpacking this:
   1. H(m) is the double SHA256 hash of message m
   2. Treating the SHA256 hash as an integer, we add S x r which is the private key multipled by the ephemeral key's x co-ordinate
   3. We multiply the sum from step 2 by k^-1
   4. We perform a modulo function with n as the denominator and the result of step 3 as numerator to get the signature

The full signature is a concatenation of r and s in DER format as outlined below:

Data Structure	Length	Data (hex)
Sequence Identifier	1	30
Length of Sequence	1	46
Integer Identifier	1	02
Byte-length of r	1	20 or 21 (decimal 32 or 33)
r (must be positive)	32 if positive or if negative, append 0x00 in MS byte to generate absolute value giving length 33 bytes	e.g. e9d34347e597e8b335745c6 53580f4cbdb4bcde2794ef7a 15d996642
Integer identifier	1	02
Byte-length of s	1	20 or 21 (decimal 32 or 33)
s (must be positive)	32 if positive or if negative, append 0x00 in MS byte to generate absolute value giving length 33 bytes	e.g. df2ccb52c7243c55bde34934 55efbdac21c74a20bb7b438c 6de3311f
Sighash type	1	e.g 41 (SIGHASH_ALL)

Insert chapter 4 animation 6 again

Chapter 5 video 2

Script	Video
Regarding the core technologies in Bitcoin, the Elliptic Curve Digital Signature Algorithm (ECDSA) stands as a relatively recent innovation in computer science	Todd
Elliptic curve signatures use the properties of the secp256k1 Elliptic curve within a massive range offering extensive applicability across a broad range of scenarios.	Show the secp256k1 curve zooming out to shov scale. see https://wiki.bitcoinsv.io/index.php/Secp256k1
Digital signatures find widespread use across various applications	Todd
and are legally recognized in most jurisdictions as equivalent to ink signatures on paper.	show someone signing paper
Digital signatures are generated using the private key of a public/private key pair and a specific message.	Show a digital signature being generated
To validate a digital signature, the validator requires both the message and the corresponding public key.	Show a digital signature being checked against message and the key
In the context of a Bitcoin transaction, the message used digital signatures	Todd
is constructed using components of the transaction and the outputs being spent as inputs.	Show a digital signature being made from a transaction
This design ensures that nodes can validate messages without relying on external information beyond the transaction itself.	Todd
Bitcoin employs digital signatures	Todd
formatted in an extended version of the DER signature format.	Show DER packet - see chapter 4 animation 6
This extended format includes a Sighash flag, granting the signing party precise control over which parts of the transaction are being signed.	animation based on https://wiki.bitcoinsv.io/index.php/SIGHASH_flag
Such granular control holds significance within a legal context.	Todd
Digital signatures serve as a fundamental element of Bitcoin, and this chapter will showcase how	Todd

these constructs can be utilized to create Turing-complete applications.	

Chapter 5 animation 1

Show an animation of the following elements being assembled into a signature message

Item	Name	Description, Length and format
1	nVersion	The version of the protocol being used to define this transaction - 4 Byte little-endian integer
2	hashPrevouts	either a SHA256 hash of the transaction's concatenated input source / sources or a 32 byte null string if SIGHASH_ANYONECANPAY is used 32 Byte hash string
3	hashSequence	the concatenation of the input nSequence value / values or a null string SIGHASH_ANYONECANPAY is used - 32 Byte hash string
4	outpoint	Outpoint for input being signed - concatenation of input TXID + Vout - 32 Byte transaction hash + 4 Byte little-endian integer
5	scriptLen	Locking script length - a VarInt - 1, 3, 5 or 9 bytes depending on script length
6	scriptPubKey	Locking script for output being spent - scriptLen byte string
7	value	Value of input - An integer representing the input's satoshi value - 8 Byte little-endian integer - 4 byte little-endian integer
8	nSequence	Then nSequence value for input being signed - 4 byte little-endian integ
9	hashOutputs	If SIGHASH_ALL is used, this value is the SHA256 hash of the concatenation of all transaction outputs in Transaction output format. If SIGHASH_SINGLE is used, this value is the SHA256 hash of only the output corresponding to the input's Vin index. If SIGHASH_NONE is use this is a null string - 32 Byte hash string
10	nLocktime	The nLocktime value for the transaction - 4 byte little-endian integer
11	sighash	The SIGHASH flags being applied to this signature - 4 bytes, XX000000 where XX is Sighash Flag

Chapter 5 assessment 1

03 - OP_PUSH_TX

OP_PUSH_TX is a scripting technique where the transaction script requires the user to submit a transaction pre-image as part of the solution.

Within the script, the pre-image is signed and then checked using one of Bitcoin's CHECKSIG opcodes (e.g. OP_CHECKSIG, OP_CHECKSIGVERIFY).

The SIGHASH flags applied to the signature can give you a means to check things such as total quantity of inputs and outputs, output script types and more. More detail on Sighash flags can be found HERE.

There are different versions of the `OP_PUSH_TX` technique, but for the purposes of this module, we will use a simplified version known as 'Optimised OP_PUSH_TX'.

To simplify the calculation process, Optimised OP_PUSH_TX uses pre-set values for both the private key and ephemeral key, allowing OP_PUSH_TX to be executed with a script of less than 100 bytes.

Insert Chapter 5 animation 2

Chapter 5 animation 2

04 - Signing and Checking the Pre-Image

Optimal OP_PUSH_TX

The following script called 'Optimal OP_PUSH_TX' has been developed between a group of companies including nChain and sCrypt for use on the BitcoinSV ledger. It is a heavily optimised OP_PUSH_TX script which uses less than 100 bytes of script.

To perform the OP_PUSH_TX validation, the pre-image is duplicated and a digital signature created from its hash. The script then checks the signature to validate the pre-image.

The script to perform the `OP_PUSH_TX` validation is as follows:

Stack	Script	Description
<tx_preimg>	OP_DUP	Duplicate the pre-image
<tx_preimg> <tx_preimg>	OP_HASH256 (double SHA256 hash	Double SHA256 hash of pre-imag
<tx_preimg> (<tx_preimg>)+00	OP_BIN2NUM	Re-encode as an optimal integer (strips leading zeroes)
<tx_preimg> optimal_dh(tx_preimg))	OP_1ADD	Add 1 to generate the uncertain signature
<tx_preimg> uncertain_length_signature	0x20	add 32 to the stack
<tx_preimg> uncertain_length_signature 0x20	OP_NUM2BIN	Reset length to 32 bytes
<tx_preimg> <signature>	3044022079be667ef9dc bbac55a06295ce870b07 029bfcdb2dce28d959f28 15b16f817980220	Add the DER integer prefix, knowr value and signature length to the stack
<tx_preimg> <signature> <prefix+r>	OP_SWAP	Swap into correct order
<tx_preimg> <prefix+R> <signature>	OP_CAT	Join prefix and R to signature
<tx_preimg> <DER_Signature>	0x41	Add SIGHASH flag to stack (e.g. `SIGHASH_ALL`)
<tx_preimg> <DER_Signature> 0x41	OP_CAT	Finalise signature
<tx_preimg> <signature>	02b405d7f0322a89d0f9f 3a98e6f938fdc1c969a8d 1382a2bf66a71ae74a1e 83b0	Add the pubkey to the stack
<tx_preimg> <signature> <pubkey>	OP_CHECKSIGVERIFY	Check the signature is valid
<tx_preimg>	...	Pre-image has been validated

Insert Chapter 5 animation 2

What's happening here?

The double hash of the pre-image is created using the `OP_HASH256` opcode. We then use 33 OP_NUM2BIN to extend it to 33 bytes long to ensure it will be recognised as a positive value. We then add 1 to the big endian value which is the same as adding 2^256 to the value, which is the value of our private key. We then use 0x20 OP_NUM2BIN to extend the final big-endian signature to exactly 32 bytes.

Following this, the DER sequence of integer identifiers, lengths and the R-value are attached to the beginning.

Finally the SIGHASH flag is added (in this case, SIGHASH_ALL). See https://wiki.bitcoinsv.io/index.php/SIGHASH_flags for more information on SIGHASH flags.

Once the signature is complete and on the stack, the public key is added.

Now that the full signature and public key are both on the stack OP_CHECKSIGVERIFY can be used to check that the transaction pre-image was correct and that the values it contains match the parameters of the transaction, thereby completing the OP_PUSH_TX operation.

Accomodating the Low S Rule

Thanks to the nature of Elliptic Curve digital signatures, each combination of message hash, private key and ephemeral key can produce 2 valid signatures. One is a positive number and one is a negative number expressed using 2's complement. Nodes on the BitcoinSV ledger currently require that the signature portion of a DER encoded signature be the positive or 'Low' signature value. Unfortunately, it is not possible to tell which of the two values will be output by the equation meaning that the script will fail 50% of the time. To ameliorate this, applications that leverage the Optimal OP_PUSH_TX script must be written to malleate some part of the transaction until a valid pre-image that solves to a Low S value is found. Typically this is done by changing some part of an output or the nLocktime value.

Classic OP_PUSH_TX

An alternative to Optimal OP_PUSH_TX is Classic OP_PUSH_TX. Classic OP_PUSH_TX has a much lower failure rate which is achieved by changing the endianness of the signature and ensuring that it is 'low S' rather than high S. This method has just a 1/256 failure rate making it more robust than the 'Optimal' version. The script is as follows:

```
OP_HASH256 OP_DUP OP_FALSE OP_GREATERTHAN OP_IF OP_1ADD OP_ELSE OP_1SUB
OP_ENDIF 0x20 OP_NUM2BIN OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT
OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_DUP OP_FALSE OP_LESSTHAN OP_IF
414136d08c5ed2bf3ba048afe6dcaebafeffffffffffffffffffffffffffffff OP_SUB
OP_ENDIF OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE
OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_TRUE OP_SPLIT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_SWAP OP_CAT
3044022079be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798022
0 OP_SWAP OP_CAT 0x41 OP_CAT
02b405d7f0322a89d0f9f3a98e6f938fdc1c969a8d1382a2bf66a71ae74a1e83b0
OP_CHECKSIGVERIFY
```

Other versions of OP_PUSH_TX

Other versions of OP_PUSH_TX have been implemented which allow the user to submit their own private key, their own K-values and much more. Each has its own advantage and disadvantage in terms of its length and failure rate. The following paper from nChain contains several alternate examples.

INSERT nChain paper.

For the purposes of the remainder of this course, we will use the Optimal OP_PUSH_TX script for brevity.

Chapter 5 animation 3

Animate the signature and checking process

Chapter 5 assessment 2

05 - nVersion

nVersion is the first parameter in the transaction pre-image. It's value indicates the version of the node client that should be used to evaluate this transaction (4 bytes)

nVersion can be evaluated by splitting the first 4 bytes of the pre-image and moving the value to the top of the stack. In this way, a user can pre-set a version flag and enforce its use in a future transaction that spends the output.

Insert Chapter 5 animation 4

This functionality was originally available in script using the `OP_VER`, `OP_VERIF` and `OP_VERNOTIF` opcodes, however these have been disabled and are no-longer part of the scripting language.

In its original form a wallet could potentially construct a transaction that only a sub-set of nodes would try to put into a block. The functionality isn't used by miners today and so application of this technique is limited.

Example 1: Replace OP_VER

To replicate the funtionality of OP_VER, the following script can be used:

Stack	Script	Description
<tx_preimg>	...	Pre-image has been validated
<tx_preimg>	OP_4	add 4 to the stack
<tx_preimg> 4	OP_SPLIT	Split version from pre-image
<version> <r_tx_preimg>	OP_SWAP	Swap version to front, or drop if no needed
<r_tx_preimg> <version>	...	Script continues

Example 2: OP_VERIF / OP_VERNOTIF

To replicate the funtionality of `OP_VERIF` or `OP_VERNOTIF`, the following script outline can be used:

Stack	Script	Description
<tx_preimg>	...	Pre-image has been validate
<tx_preimg>	OP_4	add 4 to the stack
<r_tx_preimg> 0x04	OP_SPLIT	Split version from pre-image
<version> <r_tx_preimg>	OP_SWAP / OP_NIP	Swap version to front, or drop not needed
<r_tx_preimg> <version>	<expected_version>	To test the version, add a constant to the stack.
<r_tx_preimg> <version> <expected_version>	OP_EQUAL	Equality test
<r_tx_preimg>	OP_IF / OP_NOTIF	Enter if statement path
<r_tx_preimg>	...	process
<r_tx_preimg>	OP_ELSE	or alternative path (optional)
<r_tx_preimg>	...	alternative process (optional)
<remainder_data_items>	OP_ENDIF	End process - If-structure scri may leave one or more differe data items on the stack. Thes are passed to the rest of the script.
<remainder_data_items>	...	rest of script

Chapter 5 animation 4

06 - hashPrevouts

The composition of the hash of previous outputs field (hash_prevouts) is dependent on whether the `SIGHASH_ANYONECAPAY` flag is used in the transaction pre-image. If `SIGHASH_ANYONECAPAY` is set to 0, this field is the SHA256 hash of the concatenated hash of the each outpoint being spent in this transaction, listed in 36 byte format (txid - 32bytes, vout - 4bytes). If `SIGHASH_ANYONECAPAY` is set to 1, only the outpoint that this input spends is signed, so the field is entered as a 32 byte null string.

Insert Chapter 5 animation 5

Example: Put `hash_prevouts` on the top of the stack

In this example, we bring `hash_prevouts` to the top of the stack for further processing.

Stack	Script	Description
<r_tx_preimg>	...	Version field has been remov
<r_tx_preimg>	0x20	add Hash of Previous Output length to the stack
<r_tx_preimg> 0x20	OP_SPLIT	Split version from pre-image
<hash_prevouts> <r_tx_preimg>	OP_SWAP / OP_NIP	Swap version to front, or drop not needed
<r_tx_preimg> <hash_prevouts> / <r_tx_preimg>	...	Script continues

Chapter 5 animation 5

07 - hashSequence

The hashSequence field is the hash of the concatenation of the nSequence values of all inputs being signed. The make-up of the Hash of nSequence is dependent on whether `SIGHASH_ANYONECANPAY` flag is used.

If `SIGHASH_ANYONECANPAY` is **NOT** used, the hash value is a SHA256 hash of a concatenated list of the sequence number of each of the transaction's inputs.

If `SIGHASH_ANYONECANPAY` is used, the value is a 32 byte null string.

Insert Chapter 5 animation 6

Example: Check all sequence values are final

In this example, the script checks that two inputs are used, and that the input is final. To perform the check, the hash of sequence will be split from `r_tx_preimg` and tested against a hash of the expected value.

It is assumed that `version` and `hash_prevouts` have both been removed from `r_tx_preimage`.

Stack	Script	Description
<r_tx_preimg>	...	nVersion field and hashPrevouts have bee removed
<r_tx_preimg>	0x20	add hash_nSequence length to the stack
<r_tx_preimg> 0x20	OP_SPLIT	Split hash_nSequence from pre-image
<hash_nSequence> <rr_tx_preimg>	OP_SWAP	Swap hash_nSequence to front
<rr_tx_preimg> <hash_nSequence>	0xffffffffffffffff	Expected sequence value if two final input are used
<rr_tx_preimg> <hash_nSequence> 0xffffffffffffffff	OP_SHA256	Hash sequence value
<rr_tx_preimg> <hash_nSequence> <sha256_ffffffff>	OP_EQUALVERIFY	Check it is equal - fail if not
<rr_tx_preimg>	...	rest of script

This gives us a useful tool for managing closure of payment channels, providing an optimal processing path for channels submitted with fully final inputs.

Chapter 5 animation 6

Chapter 5 assessment 3

08 - Outpoint

The outpoint for the input being signed is a 36 byte long data item made of the concatenation of the input TXID and Vout (32 + 4 bytes).

Insert Chapter 5 animation 7

It can be used for a variety of purposes including:

- Source of unique token ID
- Input randomness
- Check input sources

We will now look at examples of each of these.

Example 1: Source of unique token ID

Tokens within the system require a unique 20 byte ID. This is achieved by hashing `<outpoint>` using `OP_HASH160`.

Stack	Script	Description
<r_tx_preimg>	...	nVersion, hashPrevouts and hashSequence have been removed
<r_tx_preimg>	0x24	add outpoint length to the sta
<r_tx_preimg> 0x24	OP_SPLIT	Split outpoint from pre-image
<outpoint> <rr_tx_preimg>	OP_SWAP	Swap outpoint to front
<rr_tx_preimg> <outpoint>	OP_HASH160	Calculate txid using OP_HASH160
<rr_tx_preimg> <token_id>	...	rest of script

The token_id can now be added to an output script and enforced forward. This will be covered later in this chapter.

Example 2: Input randomness

The TXID cannot be known until the transaction is final, making it ideal as a source of input randomness. This example uses the MODULO function to implement a Rock-Paper-Scissors game.

Stack	Script	Description
<r_tx_preimg>	...	Version, hash_prevouts and hash_nSequence have been removed
<r_tx_preimg>	0x24	add outpoint length to the sta
<r_tx_preimg> 0x24	OP_SPLIT	Split outpoint from pre-image
<outpoint> <r_tx_preimg>	OP_DROP	Drop remaining pre-image
<outpoint>	OP_SHA256	Generate random output
<random_value>	OP_3	Put 3 on the stack
<random_value> 0x03	OP_MOD	Calculate modulo 2 - 1 or 0 outcome
<rem>	OP_IFDUP	Duplicates the stack if not ze
[0x00, 0x01 0x01, 0x02 0x02]	OP_NOTIF	3 possible stack states after OP_IFDUP. If 0, enter if statement path
	<rock>	Zero is rock
[0x01 , 0x02]	OP_ELSE	If not zero, duplicated outcom remains.
[0x01 , 0x02]	OP_1	Test for 1
[0x01 , 0x02] 0x01	OP_EQUAL	Equality check
<result>	OP_IF	Enter if
	<paper>	1 is paper
	OP_ELSE	Else
	<scissors>	2 is scissors
<rock/paper/scissors>	OP_ENDIF	Endif
<rock/paper/scissors>	...	rest of script

Example 3: Check input sources

This UTXO requires that it be spent in a transaction with the two outputs below it in the previous transaction. It uses hash_prevouts which must be left on the stack in an earlier section of script.

Stack	Script	Description
<hash_prevouts> <r_tx_preimg>	...	Version and hash_nSequence have bee removed and any needed conditions hav been checked.
<hash_prevouts> <r_tx_preimg>	0x24	add outpoint length to the stack
<hash_prevouts> r_tx_preimg> 0x24	OP_SPLIT	Split outpoint from pre-image
<hash_prevouts> <outpoint> <r_tx_preimg>	OP_SWAP	Swap outpoint to front
<hash_prevouts> <r_tx_preimg> <outpoint>	OP_DUP	
<hash_prevouts><r_tx_preimg> <outpoint> <outpoint>	0x20	add 32 to stack
<hash_prevouts><r_tx_preimg> <outpoint> <outpoint> 0x20	OP_SPLIT	Split the txid and vout from the outpoint
<hash_prevouts><r_tx_preimg> <outpoint> <txid><vout>	OP_1ADD	add 1 to the vout
<hash_prevouts><r_tx_preimg> <outpoint> <txid><vout+1>	OP_2DUP	Duplicate txid and vout+1
<hash_prevouts><r_tx_preimg> <outpoint> <txid><vout+1> <txid> <vout+1>	OP_1ADD	add 1 to the vout
<hash_prevouts><r_tx_preimg> <outpoint> <txid><vout+1> <txid> <vout+2>	OP_CAT	Create outpoint 3
<hash_prevouts><r_tx_preimg> <outpoint> <txid><vout+1> <outpoint3>	OP_CAT	Join to previous vout
<hash_prevouts><r_tx_preimg> <outpoint> <txid> <vout+1&outpoint3>	OP_CAT	Join to txid
<hash_prevouts><r_tx_preimg> <outpoint> <outpoint2&3>	OP_CAT	Join all three outpoints together
<hash_prevouts><r_tx_preimg> <outpoint1&2&3>	OP_SHA256	hash outpoints
<hash_prevouts><r_tx_preimg>		

<hashoutpoints>	OP_ROT	Rotate hash_prevouts to top of stack
<r_tx_preimg> <hashoutpoints> <hash_prevouts>	OP_EQUALVERIFY	Check condition is met
<r_tx_preimg>	...	Rest of script

Chapter 5 animation 7

09 - scriptLen and scriptPubKey

Part of the transaction pre-image is the scriptPubKey held in the UTXO being spent in the input. This is broken down as two fields as follows:

1. scriptLen - the locking script length, a VarInt (1, 3, 5 or 9 bytes depending on script length)
2. scriptPubKey - the locking script for the UTXO being spent (Length as defined by previous parameter)

To extract the locking script, we must first extract the length. There are 4 possible sizes for the VarInt, depending on the length of the script. The size of this field can be inferred from the value in its first byte:

- If the value is equal to or less than 0xFC, the varInt is a 1 byte integer value containing the integer value of scriptLen.
- If the first byte is 0xFD, the varInt is 3 bytes long, with the last 2 bytes containing the integer value of scriptLen.
- If the first byte is 0xFE, the varInt is 5 bytes long, with the last 4 bytes containing the integer value of scriptLen.
- If the first byte is 0xFF, the varInt is 9 bytes long, with the last 8 bytes containing the integer value of scriptLen.

Insert Chapter 5 animation 8

Example 1: 2 byte length field

Typically, we can know roughly how

big a script might be. A 2 byte length is valid from 253B up to 64kB so we can assume for our purposes this is what we are expecting.

Stack	Script	Description
<r_tx_preimg>	...	Version, hash_prevouts, hash_nSequence and hash_outpoints have been removed
<r_tx_preimg>	0P_3	VarInt is 3 bytes long
<r_tx_preimg> 0x03	OP_SPLIT	Split length
<varint> <rr_tx_preimg>	OP_SWAP	Move to top of stack
<rr_tx_preimg> <varint>	OP_1	First byte is VarInt length field
<rr_tx_preimg> <varint> 0x01	OP_SPLIT	Calculate txid using OP_HASH160
<rr_tx_preimg> <varint_id> <length_be>	OP_NIP	Nip varint ID
<rr_tx_preimg> <length_be>	OP_SPLIT	Split script from pre-image
<rrr_tx_preimg> <lock_script>	...	Rest of script

Now that the script is on the stack, it is possible for it to utilise data stored in itself to enforce the conditions of the next output state. We will get into this shortly.

Example 2: Variable VarInt handling

This example splits a script of unknown length from the stack. It first splits off the 1-byte VarInt type identifier, checks whether the varInt is 1 byte, 3 bytes or 5 bytes long, and then where needed splits the length value from r_tx_preimg before separating the script from the pre-image.

Stack	Script	Description
<r_tx_preimg>	...	Version, hash_prevouts, hash_nSequence and hash_outpoints have been removed
<r_tx_preimg>	0P_1	VarInt type ID is 1 byte
<r_tx_preimg> 0x01	OP_SPLIT	Split type
<type> <rr_tx_preimg>	OP_SWAP	Move to top of stack
<rr_tx_preimg> <type>	OP_2	
<rr_tx_preimg> <type> 0x02	OP_NUM2BIN	
<rr_tx_preimg> <type>	OP_DUP	Duplicate type
<rr_tx_preimg> <type> <type>	0xFC	0xFC or less is 1 byte varInt
<rr_tx_preimg> <type> <type+00> 0xFC00	OP_GREATERTHANOREQUALTO	Is type >= 0xFE? - Use GREATERTHANOREQUAL because 0xFE is a NEGATIV integer
<rr_tx_preimg> <type> <result>	OP_NOTIF	If NOT, enter loop. Otherwise <type> is length
<rr_tx_preimg> <type>	OP_DUP	Duplicate type
<rr_tx_preimg> <type> <type>	0xFD	Is length 2 bytes?
<rr_tx_preimg> <type> <type> 0xFD	OP_EQUAL	Test
<rr_tx_preimg> <type> <result>	OP_IF	If 2 bytes then...
<rr_tx_preimg> <type>	OP_DROP	Drop type
<rr_tx_preimg>	OP_2	2 byte length
<rr_tx_preimg> 0x02	OP_SPLIT	Split
<length> <rrr_tx_preimg>	OP_SWAP	Move length_bigendian to to stack
<rr_tx_preimg> <type>	OP_ELSE	If not 2-byte
<rr_tx_preimg> <type>	0xFE	Is length 4 bytes?
<rr_tx_preimg> <type> 0xFE	OP_EQUAL	Check equality
<rr_tx_preimg> <result>	OP_IF	Enter if statement

<rr_tx_preimg>	OP_4	Length is 4 bytes
<rr_tx_preimg> 0x04	OP_SPLIT	Split length
<length> <rrr_tx_preimg>	OP_SWAP	Swap it to front
<rr_tx_preimg>	OP_ELSE	If not 4, must be 8
<rr_tx_preimg>	OP_8	Length is 8 bytes
<rr_tx_preimg> 0x08	OP_SPLIT	Split length
<length> <rrr_tx_preimg>	OP_SWAP	Swap to front
<rrr_tx_preimg> <length>	OP_ENDIF	Exit IF loop
<rrr_tx_preimg> <length>	OP_ENDIF	Exit IF loop
<rrr_tx_preimg> <length>	0x00	Add 00 to stack (cannot use OP_FALSE)
<rrr_tx_preimg> <length> 0x00	OP_CAT	Add zeroes to the length to ensure it will be interpreted a positive integer
<rrr_tx_preimg> <length+00>	OP_BIN2NUM	Optimally encode the nubme
<r_tx_preimg> <length>	OP_SPLIT	split the script from the pre-image remainder
<lock_script> <rrrr_tx_preimg>	OP_SWAP	Swap to the front
<rrrr_tx_preimg> <lock_script>	...	Rest of script

These script elements can easily be customised to your requirements as you define your OP_PUSH_TX script. Simple checks may require scripts smaller than 253 bytes allowing these checks to be optimised as needed. Understanding the processing of the scriptLen value is an important aspect of this process. Much care must be taken to handle integer values so that they are interpreted correctly.

Chapter 5 animation 8

10 - value

The value of the input (value) is an 8 byte little endian integer representing the satoshi value of the UTXO being signed.

This can be useful when enforcing process states that require either static or dynamic satoshi values to be used or otherwise checked in each iteration of the transaction.

Insert Chapter 5 animation 9

Example: Value of input check

Checking the value can be a useful gating process to decide whether or not to end a process. In this example, the input value is checked, and if found to be less than 16 satoshis, an alternative condition is created.

Stack	Script	Description
<r_tx_preimg>	...	Version, hash_prevouts, hash_nSequence, hash_outpo and the script have been remov
<r_tx_preimg>	OP_8	length is 8 bytes long
<r_tx_preimg> 0x08	OP_SPLIT	Split value
<8byte_value> <rr_tx_preimg>	OP_SWAP	Move to top of stack
<rr_tx_preimg> <8byte_value>	OP_BIN2NUM	Optimally encode the integer value
<rr_tx_preimg> <value>	OP_16	16 satoshis is our limit
<rr_tx_preimg> <value> 16	OP_GREATERTHANOREQUAL	numeric check
<rr_tx_preimg> <result>	OP_IF	if greater than
<rr_tx_preimg>	...	process
<rr_tx_preimg>	OP_ELSE	if less than
<rr_tx_preimg>	...	alternative process
<data_items>	OP_ENDIF	Resulting outcome

Chapter 5 animation 9

11 - nSequence

The nSequence field is a 4 byte little endian number that is the sequence number of the input. This value can be used with nLocktime to put transactions into a non-final state, allowing a managed transfer of data to take place.

Insert Chapter 5 animation 10

Example: Extract and check nSequence

In this example, we will extract the nSequence value of this input and only allow the UTXO to be spent if it is final.

Stack	Script	Description
<r_tx_preimg>	...	Version, hash_prevouts, hash_nSequence, hash_outpoints, script and valu have been removed
<r_tx_preimg>	OP_4	nSequence length is 4 bytes lo
<r_tx_preimg> 0x04	OP_SPLIT	Split nSequence
<nsequence> <rr_tx_preimg>	OP_SWAP	Move to top of stack
<rr_tx_preimg> <nsequence>	0xFFFFFFFF	0xFFFFFFFF is final
<rr_tx_preimg> <nsequence> 0xFFFFFFFF	OP_EQUALVERIFY	Fail script if tx is non-final
<r_tx_preimg>	...	rest of script

Chapter 5 animation 10

Chapter 5 assessment 4

12 - hashOutputs

hashOutputs is a SHA256 hash of the concatenation of all outputs being signed in Transaction output format.

Transaction Output Format is a concatenation of the following 3 items:

1. The value of the output in Satoshis (8 bytes)
2. The length of the locking script (varInt format)
3. The locking script itself (string)

Whether the signature is applied to all, one or none of the outputs depends on the SIGHASH flags used in the signature.

If SIGHASH_ALL is used, all of the transaction outputs are concatenated and hashed together.

If SIGHASH_SINGLE is used, only the output with the same index as the input being signed is hashed. If there is no output at that index value, hashOuts is a 32 byte null string.

If SIGHASH_NONE is used, hashOuts is a 32 byte null string.

Hash_outputs is particularly important for staged processing as it allows the script to forward-enforce the conditions within the next output, which may include use of some or all of the script used in the input, or may be chosen from a variety of pre-determined script outputs.

Insert Chapter 5 animation 11

Example 1: Selectable next-state using input

Stack	Script	Description
<input> <r_tx_preimg>	...	Version, hash_prevouts, hash_nSequence, hash_outpoints, script, value ar nSequence have been remove
<input> <r_tx_preimg>	0x20	Hash_outputs is 32 bytes (0x2(
<input> <r_tx_preimg> 0x20	OP_SPLIT	Split nSequence
<input> <hash_outputs> <rr_tx_preimg>	OP_SWAP	Swap hash_outputs to top of st
<input> <rr_tx_preimg> <hash_outputs>	OP_ROT	Move input to top of stack
<rr_tx_preimg> <hash_outputs> <input>	OP_IF	True or false input
<rr_tx_preimg> <hash_outputs>	<true_case_output>	Load the hash of the true case output onto the stack
<rr_tx_preimg> <hash_outputs>	OP_ELSE	If input is false
<rr_tx_preimg> <hash_outputs>	<false_case_output>	Load the hash of the false case output onto the stack
<rr_tx_preimg> <hash_outputs> <next_state_output>	OP_ENDIF	Correct next state output is load onto stack, exit IF loop
<rr_tx_preimg> <hash_outputs> <next_state_output>	OP_EQUALVERIFY	Check that desired next state matches transaction pre-image
<rr_tx_preimg>	...	rest of script

In this example, we are splitting the output hash from the pre-image and checking it against one of two pre-defined output hashes that represent our next state based on a true/false input. These pre-defined hashes can include any number of outputs with any script needed, and can attach any valid quantity of satoshis to each output.

Much more sophisticated checks can be done, including keeping <script> on the stack and checking that the next state includes the same script elements. This can be useful if you don't know exactly what the next state will be when the output is created. For instance, this might be for a digital object with transferrable ownership. Each new owner will want to add their own keys and checks, so this must be validated with the script.

Chapter 5 animation 11

13 - nLocktime

The nLocktime value is used for creating payment channels. If a transaction has any inputs with non-final nSequence values (e.g. nSequence < 0xffffffff) and an nLocktime in the future, it is considered a payment channel, and cannot be put into a block until either the nLocktime expires, or the transaction is updated to have all final inputs.

This can be useful if an output must be created that can't be spent before a particular time. The nSequence can be checked to be non-final, and the nLocktime can be checked to be later than a pre-set time/date value.

Insert Chapter 5 animation 12

Example: Check channel status

Stack	Script	Description
<nSequence><r_tx_preimg>	...	nSequence is left on the stack. R_tx_preimg has had version, hash_prevouts, hash_nSequer hash_outpoints, script, value, nSequence and hash_outputs removed
<nSequence> <r_tx_preimg>	OP_4	nLocktime is 4 bytes
<nSequence> <r_tx_preimg> 0x04	OP_SPLIT	Split nSequence
<nSequence> <nLocktime> <rr_tx_preimg>	OP_SWAP	Swap nLocktime to top of stack
<nSequence> <rr_tx_preimg><nLocktime>	0x00	add 0x00 (push 0x00 to stack, not use OP_FALSE)
<nSequence> <rr_tx_preimg><nLocktime> 0x00	OP_CAT	add two leading zeroes as MSI so locktime will be interpreted a positive integer
<nSequence> <rr_tx_preimg> <nLocktime+00>	<expiry>	Load pre-set expiry time to stac this is in Unix epoch time, rathe than block-height.
<nSequence> <rr_tx_preimg> <nLocktime+00> <expiry>	OP_GREATERTHANOREQUAL	Check that the nLocktime is at after the pre-set expiry
<nSequence> <rr_tx_preimg> <result>	OP_VERIFY	If expiry is valid, the script will continue
<nSequence> <rr_tx_preimg>	OP_SWAP	Swap nLocktime to top of stack
<rr_tx_preimg> <nSequence>	0x00	Push 0x00 onto the stack
<rr_tx_preimg> <nSequence> 0x00	OP_CAT	Attach 00 to front of nSequence ensure it is interpreted as a positive integer by the validatio engine
<r_tx_preimg> <nSequence+00>	0xffffffff00	load finalised nSequence to sta Value is little-endian so leading zeroes/MSB are on the right

<r_tx_preimg> <nSequence+00> 0xffffffff00	OP_LESSTHAN	Check nSequence is less than final
<r_tx_preimg> <result>	OP_VERIFY	If the input is non-final, the scri can continue
<r_tx_preimg>	...	rest of script

In this script, we first check that the nLocktime is set to a time in the future, before also checking whether the input is non-final. By doing both of these things, we can validate that the transaction will remain in an open payment channel until the nLocktime has expired, replicating and extending the functionality of removed opcode OP_CHECKLOCKTIMEVERIFY.

Chapter 5 animation 12

14 - SIGHASH flags

The Sighash flags determine which parts of the transaction are being signed.

This value impacts other elements of the pre-image including hashPrevouts, hashSequence and hashOutputs. The SIGHASH flags are all contained within the last single byte of a signature, however in the pre-image they are contained within a 4 byte value, where the left-most byte is the SIGHASH flag applied to the signature, and the remaining 3 bytes are zero.

Sighash flags work as follows:

SIGHASH_ANYONECANPAY

SIGHASH_ANYONECANPAY determines which of the transaction inputs the signature covers.

If SIGHASH_ANYONECANPAY **is not** set, the signature must include all of the transaction inputs in the hash_nSequence and hash_prevouts values.

If SIGHASH_ANYONECANPAY **is** set, only the input that the signature is applied to needs to be included. Because the previous output and nSequence are handled otherwise, both hash_nSequence and hash_prevouts are set to 32byte null strings when SIGHASH_ANYONECANPAY is set. SIGHASH_ANYONECANPAY can be useful when signing an input in advance of other transaction participants, where other inputs may be unknown.

SIGHASH_ALL, SIGHASH_SINGLE, SIGHASH_NONE

Only one of SIGHASH_ALL, SIGHASH_SINGLE and SIGHASH_NONE can be used in any signature. If more than one of these flags is set, the transaction will be invalid.

If SIGHASH_ALL is used, the signature must include all of the transaction outputs when it builds the pre-image.

If SIGHASH_SINGLE is used, the signature must include only the transaction output at the same index as the input being signed in the pre-image.

If SIGHASH_NONE is used, none of the transaction's outputs are included in the pre-image.

SIGHASH_FORKID

SIGHASH_FORKID is a flag that was added to the signatures after August 1 2017 to provide replay protection against the BTC network when it was separated from Bitcoin to add segregated witness. SIGHASH_FORKID must ALWAYS be set, or the transactions will be invalid.

Insert Chapter 5 animation 13

Example: Checking SIGHASH flags

Stack	Script	Description
<r_tx_preimg>	...	Because nSequence is the fina byte value, we do not need to know what parts of r_tx_preimg remain to be able to extract it.
<r_tx_preimg>	OP_SIZE	Put the size of r_tx_preimg on t stack
<r_tx_preimg> <size>	OP_4	SIGHASH length is 4 bytes lon
<r_tx_preimg> <size> 0x04	OP_SUB	Calculate split location
<r_tx_preimg> <split_length>	OP_SPLIT	Split the pre-image
<lr_tx_preimg> <sighash>	OP_NIP	Remove remainder
<sighash>	<required_sighash>	Load the required sighash flag onto the stack as a 1 byte value
<sighash> <required_sighash>	OP_NUMEQUALVERIFY	Use numeric equality check to validate that the correct sighash used.
...	...	rest of script

In this script, we first use OP_SIZE to push the length of r_tx_preimg to the stack. By subtracting 4 from this, we can calculate the location in the remaining pre-image where the sighash element begins, and use that value to split it from r_tx_preimg. Once sighash is on the stack, it is a simple numeric equality check against a supplied sighash.

Alternative scenarios include using a bitwise mask and performing AND/OR checks against individual flags rather than enforcing the entire sighash state.

Chapter 5 animation 13

Chapter 5 assessment 5

Chapter 6: Conclusion

Conclusion

Congratulations on completing Introduction to Bitcoin Script. You are well on your way to becoming a Bitcoin Script engineer.

The techniques you have learned are key to building next generation platforms that leverage the power of Bitcoin, and will be invaluable as part of any Bitcoin script engineer's toolkit.

Insert Chapter 6 video 1

Chapter 6 video 1

Script	Video
Congratulations on completing the Introduction to Bitcoin Script course! We trust that you have found the journey worthwhile.	Todd
The realm of Bitcoin Script holds vast uncharted territories, with countless exciting techniques awaiting exploration beyond the scope of this course.	Show OP_PUSH_TX, P2PKH, etc in text
Bitcoin Script engineers are a rare breed, making those of you who choose to pursue this path and become experts true pioneers in the field.	Show engineering inventor working alone stock footage
We hope you share in our enthusiasm for the boundless possibilities and look forward to crossing paths with you in the future.	Todd
The world eagerly awaits your talent and contributions.	Fade

Final Assessment