

Bitcoin Script ENTERING THE PROGRAMMABLE ECONOMY

Stéphane Roche

ABOUT STEPHANE



2015

Start working on Bitcoin in 2015 at Ledger (hardware wallet)



2017–2018

Focus on Bitcoin technical trainings
Founder of Bitcoin Studio
Consultant at Chainsmiths



2016–2017

Work on Ethereum in 2016–2017

- Co-founded non-profit organization Asseth
- R&D on Dao1901
- Contribute to the ERC20 Consensys smart contracts
- Dether.io (15,000 ETH raised)

@janakaSteph on Twitter
rstephane@protonmail.com

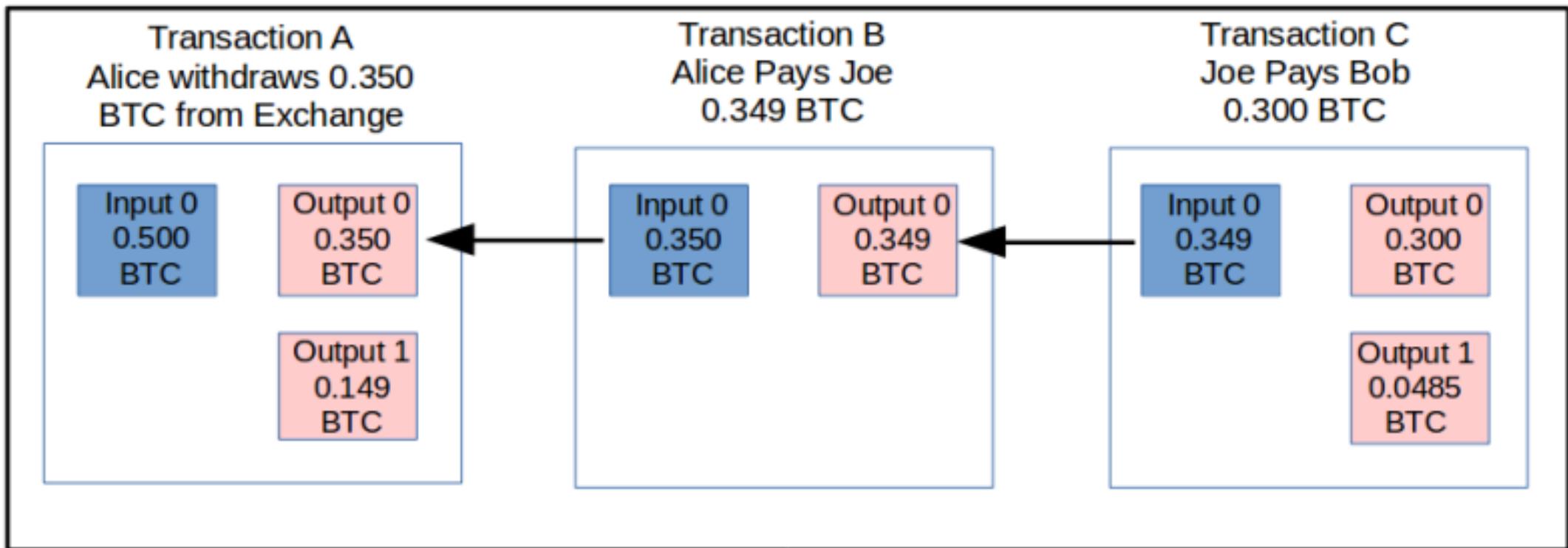
OUTLINE

- 
- 1 **Transaction Basics and Standard Output Types**
 - 2 **Script Validation Logics**
 - 3 **Opcodes**
 - 4 **Arbitrary Scripts Examples**
 - 5 **Future Improvements**



TRANSACTION BASICS AND STANDARD OUTPUT TYPES

INPUT-OUTPUT CHAIN



- Any Bitcoin transaction is technically a “smart contract”
- A Bitcoin smart contract is a predicate (returns true or false)
- Achieved through execution of challenge/response scripts
- Every bitcoin validating node executes the scripts
 - All the inputs are validated independently

Remaining Script	Stack State
<Response Script> <Challenge Script>	
<Challenge Script>	x_1 x_2 \vdots x_n
	y_1 y_2 \vdots y_m

Response is valid if top element y_1 evaluates to True

REVERSE POLISH NOTATION

- Operators follow their operands
- Commonly used in stack-oriented programming languages

Remaining Script	Stack State
OP_2 OP_3 OP_ADD	
OP_3 OP_ADD	2
OP_ADD	3 2
	5

POLICY RULES - STANDARD TX

- IsStandard() and IsStandardTx() tests
 - src/policy/policy.cpp
 - Check that tx is *standard*
 - Check various properties in inputs, outputs and other tx parts
- Only standard tx are mined and relayed by Bitcoin Core nodes
- Safety measures against DoS attacks
- Force good behavior without consensus enforcement
 - More flexible
 - Example: the tx version number

STANDARD OUTPUT TYPES

- TX_PUBKEY
- TX_PUBKEYHASH
- TX_SCRIPTHASH
- TX_MULTISIG (Bare multisig – BIP11)
- TX_NULL_DATA
- TX_WITNESS_V0_KEYHASH
- TX_WITNESS_V0_SCRIPTHASH
- TX_WITNESS_UNKNOWN
- TX_NONSTANDARD

2

SCRIPT VALIDATION LOGICS

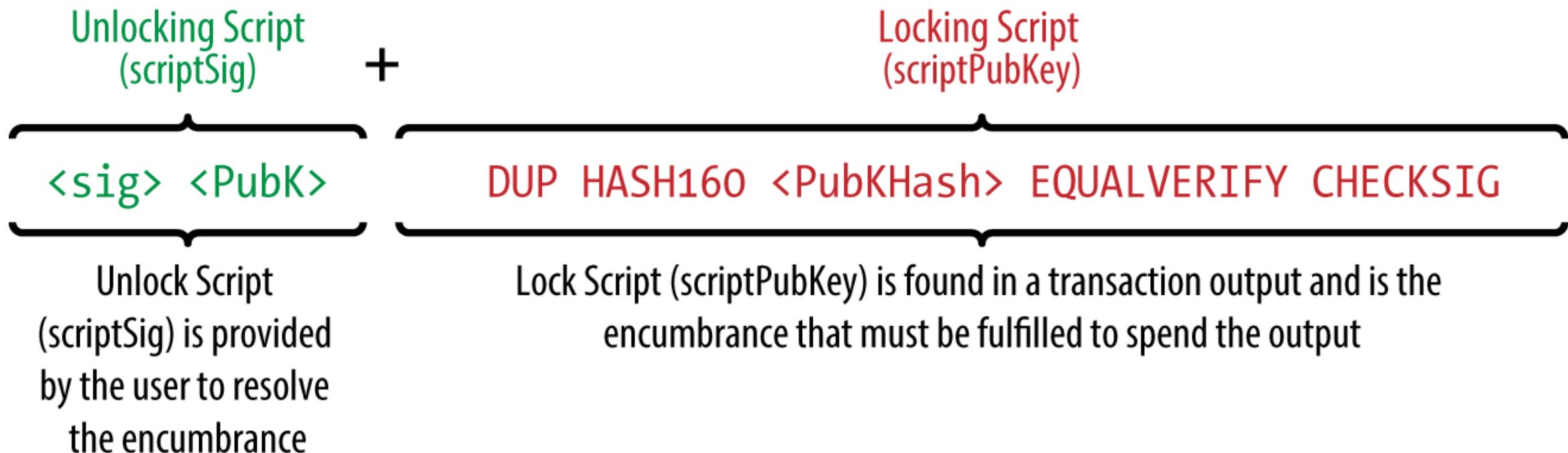
PAY TO PUBLIC KEY

- Challenge script: <Public Key> OP_CHECKSIG
- Response script: <Signature>

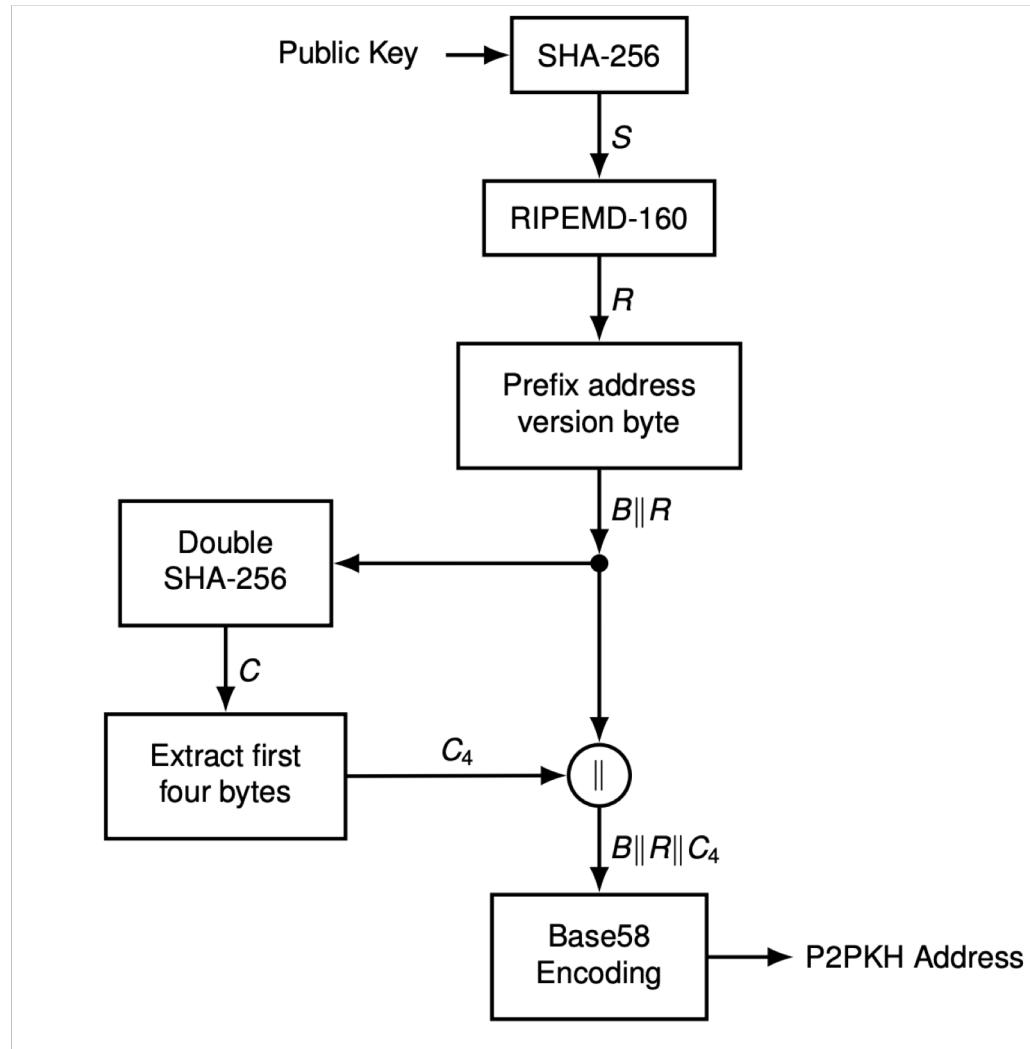
Remaining Script	Stack State
<Signature> <Public Key> OP_CHECKSIG	
<Public Key> OP_CHECKSIG	<Signature>
OP_CHECKSIG	<Public Key> <Signature>
	True/False

PAY TO PUBLIC KEY HASH

- P2PKH script has two required conditions
 - that the supplied public key match the public-key hash
 - that the supplied signature match that public key



P2PKH ADDRESS



MULTI-SIGNATURE SCRIPTS

- m-of-n multisig challenge script
 - n public keys up to 3 (standard policy)
 - m <Public Key 1> . . . <Public Key n> n OP_CHECKMULTISIG
- Response script provides signatures created using any m out of the n private keys
 - OP_0 <Signature 1> . . . <Signature m>

Remaining Script

OP_0 <Sig1> <Sig2> | OP_2 <PubKey1>
<PubKey2> <PubKey3> OP_3 OP_CHECKMULTISIG

Stack State

--

<Sig2>
<Sig1>
<Empty Array>

3
<PubKey3>
<PubKey2>
<PubKey1>
2
<Sig2>
<Sig1>
<Empty Array>

True/False

PAY TO SCRIPT HASH

- Allows specification of arbitrary scripts as payment destinations
- Specific two steps validation logic
- Challenge script
 - OP_HASH160 <RedeemScriptHash> OP_EQUAL
- Response script
 - <Response To Redeem Script> <Redeem Script>
- Cannot be used recursively inside the redeemScript itself
 - P2SH inside P2WSH or P2SH is invalid
 - P2WSH inside P2WSH is invalid

Remaining Script	Stack State
OP_0 <Sig1> <OP_1 <PubKey1> <PubKey2> OP_2 OP_CHECKMULTISIG> OP_HASH160 <RedeemScriptHash> OP_EQUAL	
<OP_1 <PubKey1> <PubKey2> OP_2 OP_CHECKMULTISIG> OP_HASH160 <RedeemScriptHash> OP_EQUAL	<Sig1> <Empty Array>
OP_HASH160 <RedeemScriptHash> OP_EQUAL	OP_1 <PubKey1> <PubKey2> OP_2 OP_CHECKMULTISIG <Sig1> <Empty Array>
<RedeemScriptHash> OP_EQUAL	<RedeemScriptHashCalc> <Sig1> <Empty Array>
OP_EQUAL	<RedeemScriptHash> <RedeemScriptHashCalc> <Sig1> <Empty Array>

Remaining Script

OP_1 <PubKey1> <PubKey2> OP_2 OP_CHECKMULTISIG

Stack State

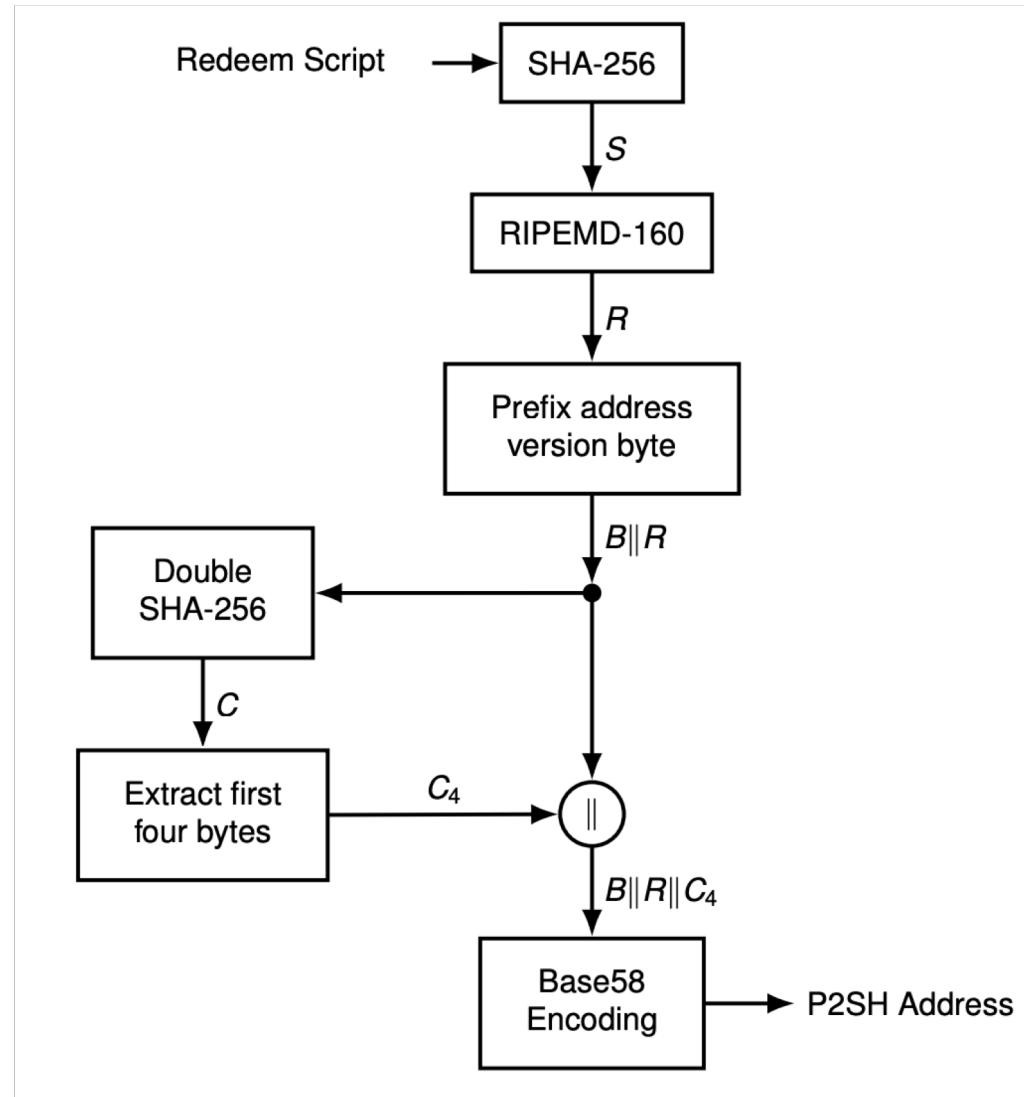
<Sig1>
<Empty Array>

OP_CHECKMULTISIG

2
<PubKey2>
<PubKey1>
1
<Sig1>
<Empty Array>

True/False

P2SH ADDRESS



NULLDATA SCRIPTS

- Challenge script: **OP_RETURN <Data>**
 - OP_RETURN terminates script execution immediately
- No valid response script exists
 - Null data outputs are unspendable
 - Any bitcoins locked by a null data challenge script are lost forever
- Policy rules
 - Maximum scriptPubkey length for the tx to be relayed is 83 bytes
 - 80 bytes of data, +1 for OP_RETURN, +2 for the pushdata opcodes
 - Only one nulldata output per tx that pays exactly 0 satoshis
- Consensus rules
 - Allow nulldata outputs up to the maximum allowed scriptPubkey size of 10,000 bytes
- Used for asset creation, document notary, digital arts and others

WITNESS VALIDATION LOGIC

- Versioned witness program triggers witness validation logic
 - <version byte> <witness program>
- Located in scriptPubkey in native witness programs
- Located in scriptSig, as a unique stack item, in P2SH witness programs

NATIVE V.0 WITNESS PROGRAMS

- **scriptSig** is empty
- **scriptPubKey** is a **versioned witness program**
 - Version byte 0 + witness program
- **Witness**
 - <signature> <pubkey> (P2WPKH)
 - data + witnessScript (P2WSH)
- **P2WPKH program**
 - 20-byte witness program must match pubKey's HASH160
 - pubKey's HASH160 and CHECKSIG are done automatically
- **P2WSH program**
 - 32-byte witness program must match witnessScript's SHA256
 - witnessScript's SHA256 and comparison is done automatically
 - The redeem script moved to witness and called witnessScript

NATIVE P2WPKH LOCKING SCRIPT

P2PKH

```
OP_DUP OP_HASH160 0067c8970e65107ffbb436a49edd8cb8eb6b567f OP_EQUALVERIFY OP_CHECKSIG
```

P2WPKH

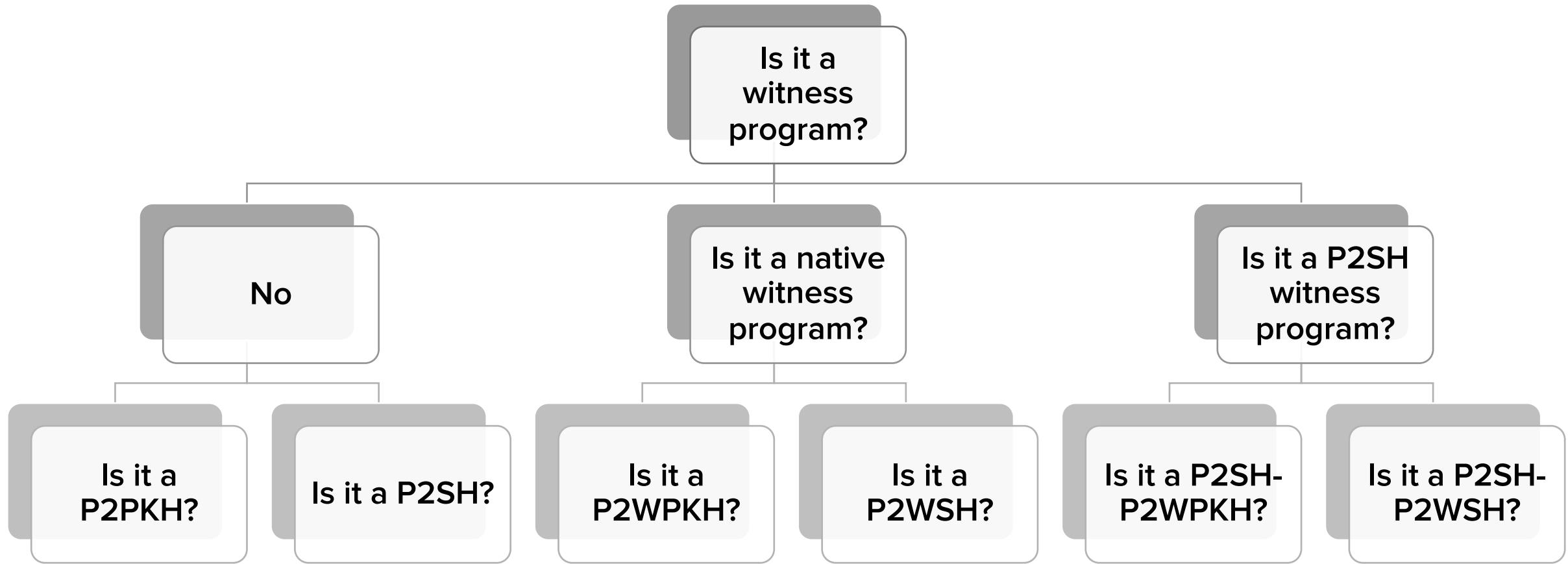
```
0 0067c8970e65107ffbb436a49edd8cb8eb6b567f
```

Witness version
/ Version byte

20-bytes witness program

P2SH V.0 WITNESS PROGRAMS

- **scriptPubkey** is a standard P2SH script
- **scriptSig** is a versioned witness program
 - VWP pushed onto the stack as a single stack item
 - HASH160
 - Hash comparison
- **Witness**
 - <signature> <pubkey> (P2SH-P2WPKH)
 - data + witnessScript (P2SH-P2WSH)
- **P2SH-P2WPKH**
 - 20-byte witness program must match pubKey's HASH160
- **P2SH-P2WSH**
 - 32-byte witness program must match witnessScript's SHA256



3

OPCODES

```
// push value
OP_0 = 0x00,
OP_FALSE = OP_0,
OP_PUSHDATA1 = 0x4c,
OP_PUSHDATA2 = 0x4d,
OP_PUSHDATA4 = 0x4e,
OP_1NEGATE = 0x4f,
OP_RESERVED = 0x50,
OP_1 = 0x51,
OP_TRUE=OP_1,
OP_2 = 0x52,
OP_3 = 0x53,
OP_4 = 0x54,
OP_5 = 0x55,
OP_6 = 0x56,
OP_7 = 0x57,
OP_8 = 0x58,
OP_9 = 0x59,
OP_10 = 0x5a,
OP_11 = 0x5b,
OP_12 = 0x5c,
OP_13 = 0x5d,
OP_14 = 0x5e,
OP_15 = 0x5f,
OP_16 = 0x60,

// control
OP_NOP = 0x61,
OP_VER = 0x62,
OP_IF = 0x63,
OP_NOTIF = 0x64,
OP_VERIF = 0x65,
OP_VERNOTIF = 0x66,
OP_ELSE = 0x67,
OP_ENDIF = 0x68,
OP_VERIFY = 0x69,
OP_RETURN = 0x6a,

// stack ops
OP_TOALTSTACK = 0x6b,
OP_FROMALTSTACK = 0x6c,
OP_2DROP = 0x6d,
OP_2DUP = 0x6e,
OP_3DUP = 0x6f,
OP_2OVER = 0x70,
OP_2ROT = 0x71,
OP_2SWAP = 0x72,
OP_IFDUP = 0x73,
OP_DEPTH = 0x74,
OP_DROP = 0x75,
OP_DUP = 0x76,
OP_NIP = 0x77,
OP_OVER = 0x78,
OP_PICK = 0x79,
OP_ROLL = 0x7a,
OP_ROT = 0x7b,
OP_SWAP = 0x7c,
OP_TUCK = 0x7d,

// splice ops
OP_CAT = 0x7e,
OP_SUBSTR = 0x7f,
OP_LEFT = 0x80,
OP_RIGHT = 0x81,
OP_SIZE = 0x82,

// bit logic
OP_INVERT = 0x83,
OP_AND = 0x84,
OP_OR = 0x85,
OP_XOR = 0x86,
OP_EQUAL = 0x87,
OP_EQUALVERIFY = 0x88,
OP_RESERVED1 = 0x89,
OP_RESERVED2 = 0x8a,

// numeric
OP_1ADD = 0x8b,
OP_1SUB = 0x8c,
OP_2MUL = 0x8d,
OP_2DIV = 0x8e,
OP_NEGATE = 0x8f,
OP_ABS = 0x90,
OP_NOT = 0x91,
OP_0NOTEQUAL = 0x92,
OP_ADD = 0x93,
OP_SUB = 0x94,
OP_MUL = 0x95,
OP_DIV = 0x96,
OP_MOD = 0x97,
OP_LSHIFT = 0x98,
OP_RSHIFT = 0x99,
OP_BOOLAND = 0x9a,
OP_BOOLOR = 0x9b,
OP_NUMEQUAL = 0x9c,
OP_NUMEQUALVERIFY = 0x9d,
OP_NUMNOTEQUAL = 0x9e,
OP_LESSTHAN = 0x9f,
OP_GREATERTHAN = 0xa0,
OP_LESSTHANOREQUAL = 0xa1,
OP_GREATERTHANOREQUAL = 0xa2,
OP_MIN = 0xa3,
OP_MAX = 0xa4,
OP_WITHIN = 0xa5,

// crypto
OP_RIPEMD160 = 0xa6,
OP_SHA1 = 0xa7,
OP_SHA256 = 0xa8,
OP_HASH160 = 0xa9,
OP_HASH256 = 0xaa,
OP_CODESEPARATOR = 0xab,
OP_CHECKSIG = 0xac,
OP_CHECKSIGVERIFY = 0xad,
OP_CHECKMULTISIG = 0xae,
OP_CHECKMULTISIGVERIFY = 0xaf,

// expansion
OP_NOP1 = 0xb0,
OP_CHECKLOCKTIMEVERIFY = 0xb1,
OP_NOP2 = OP_CHECKLOCKTIMEVERIFY,
OP_CHECKSEQUENCEVERIFY = 0xb2,
OP_NOP3 = OP_CHECKSEQUENCEVERIFY,
OP_NOP4 = 0xb3,
OP_NOP5 = 0xb4,
OP_NOP6 = 0xb5,
OP_NOP7 = 0xb6,
OP_NOP8 = 0xb7,
OP_NOP9 = 0xb8,
OP_NOP10 = 0xb9,
OP_INVALIDOPCODE = 0xff,
```

DISABLED OPCODES

- Bitwise logic operators
 - OP_INVERT, OP_AND, OP_OR, OP_XOR
- Arithmetic
 - OP_2MUL, OP_2DIV, OP_MUL, OP_DIV, OP_MOD, OP_LSHIFT, OP_RSHIFT
- String operators
 - OP_CAT, OP_SUBSTR, OP_LEFT, OP_RIGHT
- Reenabling them in legacy script execution context requires a hard fork
 - It makes previously invalid blocks valid
- Reenabling them in a new segwit script version can be done with a SF

DATA PUSH

- Direct push for short data up to 75 bytes (01 - 4b)
 - The opcode itself is the length in bytes
 - Often written as OP_PUSHBYTES in explorers
- OP_PUSHDATA1 for 8-bit values (0 to 255)
 - 4c + next byte contains byte length of data to be pushed
- OP_PUSHDATA2 for 16-bit values (0 to 65 535)
 - 4d + next two bytes contains byte length of data to be pushed
- OP_PUSHDATA4 for 32-bit values (0 to 4 294 967 296)
 - 4e + next four bytes contains byte length of data to be pushed
 - Allows pushing up to 4GB onto the stack
 - But no real use because of 520 bytes data push limit policy
- Minimal push policy
 - Only use OP_PUSHDATA1 when direct push is not possible
 - Only use OP_PUSHDATA2 when an OP_PUSHDATA1 is not possible, etc.

OP_VERIFY

- VERIFY is a conditional operator
- Pops the top item on the stack and sees if it's true; if not *it ends execution of the script*
- VERIFY is usually incorporated into other opcodes
 - OP_EQUALVERIFY, OP_CHECKLOCKTIMEVERIFY, OP_CHECKSEQUENCEVERIFY, OP_NUMEQUALVERIFY, OP_CHECKSIGVERIFY, OP_CHECKMULTISIGVERIFY
 - Each of these opcodes does its core action and then does a verify afterward
- This is how we check conditions that are absolutely required for a script to succeed

IF / THEN

- OP_IF, OP_ELSE, OP_ENDIF
- OP_NOTIF, OP_ELSE, OP_ENDIF
- OP_IFDUP
 - Duplicates the top stack item only if it's not 0
- IF conditional checks the truth of what's *before it* (top item on the stack)
- IF conditional tends to be in the locking script and what it's checking tends to be in the unlocking script

OP_CHECKLOCKTIMEVERIFY

- Absolute timelocking of UTXO
- Blockheight < 500 million >= timestamp
- 1495652013 OP_CHECKLOCKTIMEVERIFY
 - Check against May 24, 2017
- The opcode actually use the nLocktime field for consensus enforcement
 - So when respending a UTXO with CLTV, we must set the nLocktime to enable the tx

OP_CHECKSEQUENCEVERIFY

- Relative timelocking of UTXO
- 100 OP_CHECKSEQUENCEVERIFY
 - UTXO held for a hundred blocks past its mining
- 4224679 OP_CHECKSEQUENCEVERIFY
 - 6 months encoded according to BIP68
 - Multiple of 512 seconds + 23rd bit to true (here in decimal)
- The opcode actually use the nSequence field for consensus enforcement
 - So when responding a UTXO with CSV, we must set the nSequence to enable the tx
- Used in Lightning Network to chain transactions
 - A child tx cannot be used until the parent tx has been propagated, mined, and aged by the time specified in the relative timelock

ALTSTACK

- OP_TOALTSTACK, OP_FROMALTSTACK
- Common feature in stack-based languages (cf. Forth)
- Not used in practice
- We can avoid using OP_(TO|FROM)ALTSTACK by putting things onto the stack in a different order
 - There are 18 stack manipulation operators, but only OP_DUP is used with any regularity

4

SCRIPTS EXAMPLES

POOR MAN'S 1 OF 2 MULTISIG

IF

 OP_DUP
 OP_HASH160
 OP_PUSHBYTES_20 <pubKeyHashA>

ELSE

 OP_DUP
 OP_HASH160
 OP_PUSHBYTES_20 <pubKeyHashB>

ENDIF

OP_EQUALVERIFY
OP_CHECKSIG

- Alice unlocking script
 - <signatureA> <pubKeyA> True
- Bob unlocking script
 - <signatureB> <pubKeyB> False

POOR MAN'S 1 OF 2 MULTISIG #2

OP_DUP OP_HASH160 <pubKeyHashA> OP_EQUAL

IF

 OP_CHECKSIG

ELSE

 OP_DUP OP_HASH160 <pubKeyHashB> OP_EQUALVERIFY OP_CHECKSIG

ENDIF

- Alice unlocking script
 - <signatureA> <pubKeyA>
- Bob unlocking script
 - <signatureB> <pubKeyB>

ALGEBRA PUZZLES

- $x + y = 99$
 - OP_ADD 99 OP_EQUAL
 - 98 1
- $3x + 7 = 13$
 - OP_DUP OP_DUP 7 OP_ADD OP_ADD OP_ADD 13 OP_EQUAL
 - 2
- $x + y = 3, y + z = 5, x + z = 4$
 - OP_3DUP OP_ADD 5 OP_EQUALVERIFY OP_ADD 4 OP_EQUALVERIFY OP_ADD 3 OP_EQUAL
 - 1 2 3

COMPUTATIONAL PUZZLES

- Crowdsourcing a computation
 - Script requires the answer to computation, fund the P2SH as a reward
- Peter Todd's hash collision bounties
 - <value1> <value2>
 - OP_2DUP OP_EQUAL OP_NOT OP_VERIFY OP_SHA1 OP_SWAP OP_SHA1 OP_EQUAL
 - When SHA-1 was broken, 2.48 BTC were claimed

HASHLOCK

- Restricts the spending of an output until a specified piece of data is publicly revealed
- We can create multiple outputs all restricted by the same hashlock
- OP_HASH256 6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d619000000000000 OP_EQUAL
 - Solution is the genesis block header
- No signature, so not secure

- Hashlock enables payment relay
 - Allows to bind two otherwise unrelated transactions together
- Alice wants to pay Carol using Bob as an intermediary
 - Carol produces a hash from a secret s
 - Gives the hash to Alice
 - Alice pays Bob with his sig + hash
 - Bob pays Carol with her sig + hash
 - Spending Bob's payment requires Carol to publish s
 - Also allowing Bob to spend Alice's payment
- Payment relay of this sort is both contrived and insecure
 - But groundwork for much more robust protocols

HASHED TIMELOCK CONTRACT

- General mechanism for off-chain contract negotiation
 - Secret can be presented within an invalidation time window
 - Sharing the secret guarantee to the counterparty that the transaction will never be broadcast

```
HASH160 DUP <R-HASH> EQUAL
IF
    "24h" CHECKSEQUENCEVERIFY
    2DROP
    <Alice's pubkey>
ELSE
    <Commit-Revocation-Hash> EQUAL
    NOTIF
        "2015/10/20 10:33" CHECKLOCKTIMEVERIFY DROP
    ENDIF
    <Bob's pubkey>
ENDIF
CHECKSIG
```

5

FUTURE IMPROVEMENTS

ELEMENTS

- Can operate as a standalone blockchain or as a pegged sidechain
- Advanced features extending the Bitcoin protocol
- Includes several new script opcodes
 - Reintroduces most disabled opcodes
 - OP_DETERMINISTICRANDOM produces a random num within a range from a seed
 - OP_CHECKSIGFROMSTACK verifies a signature against a message on the stack
- Launched sidechains
 - Elements Alpha: Bitcoin's testnet sidechain launched in 2015
 - Liquid: Bitcoin's mainnet sidechain launched in 2018

CHECKSIGFROMSTACK

- Push signed msg from script to the stack, and check that it verifies
- Some use cases
 - Create a new type of lightning channel similar to Eltoo but better
 - Oracles
 - Delegation of authorisation to spend an output
 - Covenants (with OP_CAT)
 - Secure multiparty computations
- Hopefully shipped on late 2019 Soft Fork

COVENANTS

- Restricts how funds are allowed to be spent
- Reverse covenants (input restrictions)
 - An input can only be created with this other one
 - An input can only be created if this other one doesn't exist
- Can be recursive, applying to a chain of tx
- Allows covenant vaults (E.G. Sirer)
 - Can revert a fraudulent transaction
 - Can burn hacked coins
 - Can't pay a merchant with a vault payment

SECURE MULTIPARTY COMPUTATION

- Lottery protocols that ensure that any party that aborts after learning the outcome pays a monetary penalty to all other parties

SIMPLICITY

- Bitcoin Script replacement
 - Thanks to Segwit script versioning
 - More expressive and ultra safe
 - Paper from Dr. Russell O'Connor of Blockstream in 2017
- Typed, combinator-based, functional, without recursion, sequent-calculus-based, formal denotational semantics in Coq, MAST-native
- Allows static analysis
 - Compiles to a low-level model (the Bit Machine)
 - Useful to measure the amount of computation of a script
- First step is to implement it in Elements
- Higher-level languages that compile down to Simplicity is possible, not the hard part

SCRIPT SYSTEM GOALS

- Privacy
- Space efficiency
- Computational efficiency
- We want to convince the network that what we are trying to do is authorized
 - Today, every full node validate every transactions
 - Why not just proving correct execution?
- Execution vs verifiability
- Ultimate goal is a Zero-Knowledge proof system

MERKLE BRANCHES IN SCRIPT (MAST)

- **BIP114** - Merklized Abstract Syntax Trees (Merkle tree + AST)
 - AST allows to split a program into its individual parts
- **BIP116 / BIP 117** - MAST constructs without AST
- Usually scripts are just an OR of a few keys, timelocks and hashlocks
- Why reveal all possibilities?
 - Put all disjunctions in a Merkle tree
 - Only reveal the actually taken branch
- More privacy, more storage and computational efficiency

SCHNORR-BASED CONTRACTS

- Schnorr signatures are linear, not ECDSA
 - We can add and subtract signatures
- Scriptless scripts
 - A way to do alchemy with signatures
 - Smart contracts executed off-chain, only by the parties involved
 - A valid transaction has a signature that proves correct contract execution
- Discreet log contracts
 - A way to do alchemy with public keys
 - An oracle determines division of funds
- Atomic coinswap (Adam Gibson), etc.

CONCLUSION

- Few building blocks are enough to create interesting financial smart contracts and second layer networks
- Script versioning is awesome
- We are aiming towards a verification system, less an execution platform
 - On-chain storage/execution inherently doesn't scale
 - EC Schnorr will enable this paradigm shift
- Bitcoin future is bright, BUIDL