Bitcoin and Cryptocurrency Technologies
Lecture 5: Bitcoin Transactions

Yuri Zhykin

Mar 6, 2025

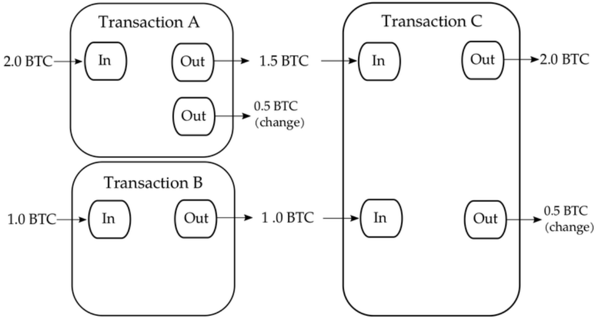
Transaction Structure

- **Tx**
  - **inputs** (ordered list of **Input**s)
  - **outputs** (ordered list of **Output**s)
- **Input**
  - **previous-tx-id** (Tx ID)
  - **previous-tx-index** (integer)
  - **unlock-script** (program)
- **Output**
  - **amount** (integer)
  - **lock-script** (program)

Transfer of Ownership 1/2

- Unspent transaction outputs (**UTXOs**) are records of bitcoin ownership - bitcoin is locked to owners via lock-scripts.
- Bitcoin transactions transfer bitcoin by *destroying* subsets of all unspent *outputs* (by providing *inputs* that unlock the output scripts) and creating new unspent *outputs*.
- The set of all *UTXOs* represents all bitcoin currently in circulation.

Transfer of Ownership 2/2



Transaction Validation

1. **Double spend check**: check that outputs referenced by inputs have not been spent yet.
2. **Inflation check**: check that the transaction does not create new base bitcoin units.
3. **Contract validation**: execute lock- and unlock-scripts.

Bitcoin Script

- **Bitcoin Script** or simply **Script** is a **stack-based Forth-like Turing-incomplete** language for expressing locking/unlocking logic in Bitcoin transactions.
- Script provides flexibility in defining the conditions for spending each particular "chunk" of bitcoin.
- *Proof-of-Work* system provides **decentralized double spend protection**.
- *Bitcoin Script* system provides **programmability** (**smart contracts**).

Stack-based Programming

- **Stack-based programming** is a programming paradigm which relies on a **stack machine model** for passing parameters.
- Example:

1.	Code	`3 5 add 3 mul ;`
	Data	`;`
2.	Code	`5 add 3 mul ;`
	Data	`3;`
3.	Code	`add 3 mul ;`
	Data	`5 3;`
4.	Code	`3 mul ;`
	Data	`8;`
5.	Code	`mul ;`
	Data	`3 8;`
6.	Code	`;`
	Data	`24;`

Turing-incompleteness

- *Script* is *intentionally* Turing-incomplete.
- On of the core component of modern programming languages is missing: **loop**.
- Scripts in transactions are executed by every validating node on the network, so loops could be used as means of DoS-attacking the network.
- Loops introduce complexity that is hard to analyse statically (i.e. by "looking" at the code without executing it).
- Ethereum network uses a Turing-complete language called **Solidity**.

Bitcoin Script Operations 1/3

- *Script interpreter* consists of a stack of commands and a stack of data.
- For each input in a transaction, it's unlock-script is executed first, then the resulting stack is used to execute the lock-script of the corresponding output:
  - initialize an empty stack $Stack_0 = Stack_{empty}$
  - execute the Input's unlock-script on $Stack_0$:

$$Stack_1 = Execute(Script_{Unlock}, Stack_0)$$

  - execute corresponding Output's lock-script on $Stack_1$:

$$Stack_2 = Execute(Script_{Lock}, Stack_1)$$

  - verify that the top of the $Stack_2$ is *True*.

Bitcoin Script Operations 2/3

- Values on the data stack are byte vectors, but they can be interpreted as numbers when needed.
- *False* value is represented by a number 0, which in turn is represented either by an empty byte vector or by singleton vector $[0x80]$.
- Any value that is not *False* is considered *True*.
- Any value other than $\{[], [0x80]\}$ at the top of the stack after script execution means that the transaction contract is valid.
- Script execution can also fail, which is equivalent to immediately returning *False*.

Bitcoin Script Operations 3/3

- **constants** - adding data to the stack
- **logic and arithmetic**
- **stack manipulation** - drop, copy, etc
- **flow control** - branching, and
  - `OP_VERIFY` - fail if top of the stack is not *True*
  - `OP_RETURN` - fail; used to attach data to transactions
- **cryptography** - cryptographic operations (hash functions)
  - `OP_CHECKSIG` - check signature against a public key
  - `OP_CHECKMULTISIG` - check multiple signature against multiple public keys (N/M signature mechanism)
- **locktime** - locktime and sequence verification

Standard Scripts 1/4

- **P2PKH** - pay-to-pubkey-hash

  Lock `OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG;`

  Unlock `<sig> <pubKey>;`

- Executing P2PKH unlock-script

1. Code `<sig> <pubKey>;`
   Data `;`
2. Code `<pubKey>;`
   Data `<sig>;`
3. Code `;`
   Data `<pubKey> <sig>;`

Standard Scripts 2/4

- Executing P2PKH lock-script

1.	Code	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG;
	Data	<pubKey> <sig>;
2.	Code	OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG;
	Data	<pubKey> <pubKey> <sig>;
3.	Code	<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG;
	Data	<pubKeyHash> <pubKey> <sig>;
4.	Code	OP_EQUALVERIFY OP_CHECKSIG;
	Data	<pubKeyHash> <pubKeyHash> <pubKey> <sig>;
5.	Code	OP_CHECKSIG;
	Data	<pubKey> <sig>;
6.	Code	;
	Data	True;

Standard Scripts 3/4

- **P2PK** - pay-to-pubkey (obsolete; reveals public key way before its corresponding private key is used to spend the output)

  Lock
  ```
  <pubKey> OP_CHECKSIG;
  ```
  Unlock
  ```
  <sig>;
  ```
- **P2MS** - M/N multisignature transaction

  Lock
  ```
  <M> <pk1> ... <pkN> <N> OP_CHECKMULTISIG;
  ```
  Unlock
  ```
  OP_0 <sig1> ... <sigM>;
  ```
- **P2SH** - pay-to-script-hash - a protocol upgrade introduced in 2012 to allow for custom lock-scripts

  Lock
  ```
  OP_HASH160 <scriptHash> OP_EQUAL;
  ```
  Unlock
  ```
  <customLockScript...> <serializedRedeemScript>;
  ```

Standard Scripts 4/4

- **P2SH** required a modification to the *Script* execution rules:
  - unlock-script is executed, resulting in `<serializedRedeemScript>` at the top of the stack
  - lock-script is executed, verifying that the `<serializedRedeemScript>` hash matches the `<scriptHash>`
  - old (non-upgraded) nodes consider transaction valid at this point
  - new (upgraded) nodes continue by deserializing the `<serializedRedeemScript>` and executing it as if it was the lock-script
- **Soft-fork tightens** the validation rules
  - non-upgraded nodes consider new data always valid, while upgraded nodes apply additional rules
- **Hard-fork relaxes** validation rules
  - non-upgraded nodes will reject new data, resulting in a network split, so all nodes must be upgraded for hard-fork to succeed

Types of Network Forks

- **Soft-fork tightens** the validation rules
  - non-upgraded nodes consider new data always valid, while upgraded nodes apply additional rules
- **Hard-fork relaxes** validation rules
  - non-upgraded nodes will reject new data, resulting in a network split, so all nodes must be upgraded for hard-fork to succeed
- **Hard-form** disconnects non-upgraded nodes from the main network.

Non-standard Scripts

- **SHA256 puzzle** - can be spent by anyone, who can provide a byte sequence s such that $h = SHA256(s)$

  Lock `OP_HASH256 <h> OP_EQUAL;`

  Unlock `<s>;`

- **SHA1 collision problem** - created by Peter Todd in 2013 to incentivize finding collisions for SHA1 hash functions, which was believed to be insecure; bounty of 2.48 Bitcoin claimed in 2017:

  Lock `OP_2DUP OP_EQUAL OP_NOT OP_VERIFY OP_SHA1 OP_SWAP OP_SHA1 OP_EQUAL;`

  Unlock `<preimage1> <preimage2>;`

Bitcoin Address 1/2

- For *standard* transactions (i.e. transactions with standard lock/unlock scripts), there is a defined "address" format.
- Bitcoin address is a short (relatively) identifier that unambiguously describes a lock-script and is used instead of providing the entire lock-script
  - for *P2PKH*, it's `<pubKeyHash>`:

$$A_{P2PKH} = Encode_{Base58Check}(HASH160(pubkey))$$

  - for *P2SH*, it's `<scriptHash>`:

$$A_{P2SH} = Encode_{Base58Check}(HASH160(redeemscript))$$

Bitcoin Address 2/2

- In order to remove any ambiguity and reduce the possibility of error, legacy Bitcoin addresses use a special **Base58Check** encoding:

$$Base58Check(t, s) = Base58(t + s + HASH256(t + s)[0:4])$$

- **Base58** encoding is similar to **base64** encoding but intentionally drops characters that can be mistaken for other characters: 0, O, I, and l.
- Value t is used to identify the type of encoded information:

0	1	"18vnsHEtfTZgaJD6Sv7QTpo2nwxkFxJgrp"	P2PKH address
5	3	"38RctgcqjcFhfGK7ynpWZQZgCTVuCoNFu"	P2SH address
111	m or n	"mkHS9ne12qx9pS9VojpwU5xtRd4T7X7ZUt"	Testnet P2PKH address
196	2	"2N4DTeBWDF9yaF9TJVGcgcZDM7EQtsGwFjX"	Testnet P2SH address

Bitcoin Wallet 1/2

- Generally, wallet is any information that can be used to construct an unlock-script.
- Typical Bitcoin wallet is a piece of software that manages cryptographic keys and can construct *standard* transactions.
- When user wants to receive bitcoin to a P2PKH address, the wallet generates a new random private key p_i, computes a public key P_i from it and computes a new P2PKH address A_i as follows

$$A_i = Encode_{Base58Check}(HASH160(P_i))$$

Bitcoin Wallet 2/2

- The address A_i is then shared with the sender, whose wallet computes $H = Decode_{Base58Check}(A_i)$ and constructs a transaction that contains an output with the required amount of bitcoin and the lock-script

```
OP_DUP OP_HASH160 <H> OP_EQUALVERIFY OP_CHECKSIG;
```

- Once the transaction is published and confirmed, the receiver now "owns" that newly locked bitcoin as their wallet has a key for spending it.
- In order to spend bitcoin locked in a particular P2PKH output, wallet software finds the corresponding private key and creates an input for a transaction with an unlock-script that contains the corresponding public key and a signature.

The End

Thank you!