Bitcoin and Cryptocurrency Technologies
Lecture 5: Bitcoin Transactions

Yuri Zhykin

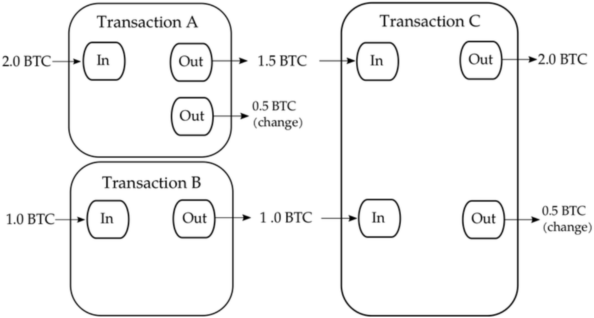
Mar 22, 2021

Transaction Structure

- Tx
  - **version**
  - **inputs** (list of **TxInputs**)
  - **outputs** (list of **TxOutputs**)
  - **witnesses** (list of **Witnesses**)
  - **locktime**
- TxInput
  - **previous-tx-id**
  - **previous-tx-index**
  - **unlock-script**
- TxOutput
  - **amount**
  - **lock-script**

Transfer of Ownership

- 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*.



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.
- Because of *Proof-of-Work*, Bitcoin is a first decentralized money system, but because of *Script* it is also the first **programmable** money system.

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 **Solidity**, which is arguably the reason behind some of the worst security incidents in the history of Ethereum.

Bitcoin Script Operations 1/3

- Script execution is the main part of transaction validation.
- *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 $S_0 = S_{empty}$
  - execute the TxInput's unlock-script on stack S_0:

$$S_1 = Execute(UnlockScript, S_0)$$

  - execute corresponding TxOutput's lock-script on stack S_1:

$$S_2 = Execute(LockScript, S_1)$$

  - verify that the top of the stack 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 [0x80] vector.
- Any value that is not *False* is considered *True*, i.e. any value other than [] or [0x80] on the top of the stack after script execution means that the transaction is valid.
- Script execution can also fail, which is equivalent to immediately returning *False* i.e. failing the transaction validation.

Bitcoin Script Operations 3/3

- Script operations are divided into the following categories:
  - **constants** - adding data to the stack
  - **flow control** - branching, and
    - `OP_VERIFY` - fail if top of the stack is not *True*
    - `OP_RETURN` - fail (used to attach data to transactions)
  - **stack manipulation** - dropping, copying, swapping elements on the stack
  - **bitwise logic and arithmetic**
  - **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 while having an address format and a size limit

  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, i.e. non-upgraded nodes consider new data always valid, while upgraded nodes apply additional rules
- **Hard-fork relaxes** validation rules, i.e. non-upgraded nodes will reject new data, resulting in a network split, so all nodes must be upgraded for hard-fork to succeed

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 s relatively short identifier that unambiguously specifies the key information in the lock-script and can be used to identify and/or reconstruct the corresponding 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, Bitcoin addresses use the 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	P2PKH address
5	3	P2SH address
111	m or n	Testnet P2PKH address
196	2	Testnet P2SH address

Bitcoin Wallet 1/2

- **So, what is a Bitcoin wallet?**
- Generally, wallet is any item that contains information that can be used to construct an unlock-script for some *UTXO*, so must be kept secret.
- Since most of transactions are *standard*, typical Bitcoin wallet is a software that can store cryptographic keys in a reasonably secure way.
- When user wants to receive bitcoin, they generate a new address, which means that wallet software 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

- This address can then be shared with the sender to send the bitcoin, which means that sender's 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/or confirmed, user "owns" that newly locked bitcoin and their wallet software can be used to spend it.
- In order to spend bitcoin locked in a particular output, wallet software finds the private key that corresponds to that address in the output, and constructs a transaction that contains the input with the unlock-script, that consists of the corresponding public key and a signature.

The End

Thank you!