IT486 v3.0

ECDSA, Bitcoin Addresses and Transactions

Recap: Public Key Cryptography

- Private key is the scalar (denoted w/lower-case letter "s")
- Public key is the resulting point sG (denoted w/upper-case letter "P")

Recap: Public Key Cryptography

- Private key is the scalar (denoted w/lower-case letter "s")
- Public key is the resulting point sG (denoted w/upper-case letter "P")
- Public key is a point (x, y) and thus has 2 numbers

SEC Format

- Public key (point on curve) serialized
- Uncompressed (65 bytes)

```
047211a824f55b505228e4c3d5194c1fcfaa15a456abdf37f9b9d97a4040afc073dee6c8906498
4f03385237d92167c13e236446b417ab79a0fcae412ae3316b77

        - 04 - Marker
        - x coordinate - 32 bytes
        - y coordinate - 32 bytes
```

SEC Format

- Public key (point on curve) serialized
- Uncompressed (65 bytes)

```
047211a824f55b505228e4c3d5194c1fcfaa15a456abdf37f9b9d97a4040afc073dee6c8906498
4f03385237d92167c13e236446b417ab79a0fcae412ae3316b77

        - 04 - Marker
        - x coordinate - 32 bytes
        - y coordinate - 32 bytes
```

- Compressed

```
0349fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278a

        - 02 if y is even, 03 if odd - Marker
        - x coordinate - 32 bytes
```

ECDSA Signature Algorithm

- Start with the hash of what you're signing (z)
- Next assume you secret is e and the public key point $P = eG$
- Get a new random number k
- Compute kG. The x-coordinate $= r$
- Compute $s = (z + re)/k$
- Signer can compute s since he has e, nobody else can compute s
- Signature is simply the pair (r, s)

ECDSA Verification Algorithm

- Start with the hash of what your'e signing (z)
- Next assume you have the public point $eG = P$
- Signature is (r, s) where $s = (z + re)/k$
- Compute $u = z/s$
- Compute $v = r/s$
- Compute $uG + vP = (z/s)G + (r/s)P = (z/s)G + (re/s)G = ((z + re)/s)G = ((z + re)k/(z + re))G = kG = (r, y)$
- If x coordinate matches r, you have a valid signature

The danger of k re-use

- Recall k is a secret used to calculate both the r and s values in the signature
- If we sign two messages with the same value k, we have:
  - $s_1 = \dfrac{(z_1 + re)}{k}$
  - $s_2 = \dfrac{(z_2 + re)}{k}$

The danger of k re-use

$$
\begin{aligned}
s_1 - s_2 &= \frac{z_1 - z_2}{k} \\
k &= \frac{z_1 - z_2}{s_1 - s_2}, \quad \text{Now we have } k!
\end{aligned}
$$

$$
\begin{aligned}
s_2 &= \frac{z_2 + re}{k} \\
e &= \frac{s_2 k - z_2}{r}, \quad \text{Now we have the private key!}
\end{aligned}
$$

Account-based Systems

- Each user has an account that holds a *balance*
- Transfers into or out of the account change the balance

Account-based Systems

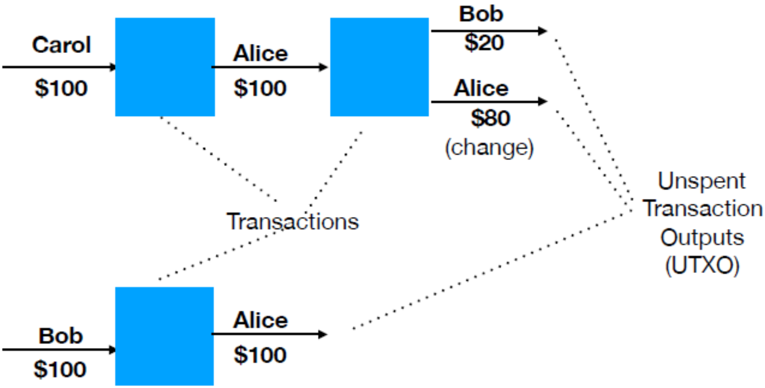
- Each user has an account that holds a *balance*
- Transfers into or out of the account change the balance
- Example
  - Alice: 0, Bob: 600, Carol: 100
    - Bob transfers 100 to Alice
    - Carol transfers 100 to Alice
  - Alice: 200, Bob: 500, Carol: 0
    - Alice transfers 20 to Bob
  - Alice: 180, Bob: 520, Carol: 0

Bitcoin Transaction Structure

- Bitcoin uses a different representation
- To send 20, Alice needs to indicate if she is spending from the 100 she got from Bob or the 100 she got from Carol

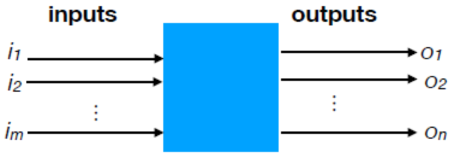
Bitcoin Transaction Structure

- When Alice spends 20 of the 100 she received from Carol, we view the situation like this:



Bitcoin Transaction Structure

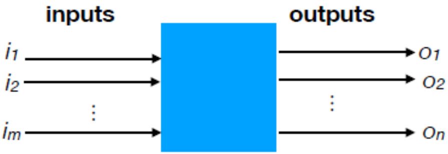
- A Bitcoin transaction looks like:



where the total amount of inputs $I = \sum_{j=1\ldots m} i_j$ and outputs $O = \sum_{j=1\ldots n} o_j$ satisfy $I \geq O$

Bitcoin Transaction Structure

- A Bitcoin transaction looks like:



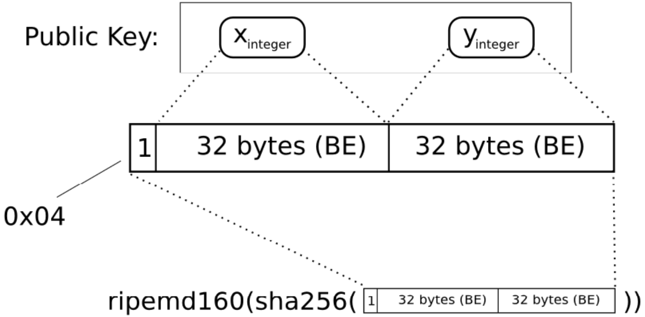
where the total amount of inputs $I = \sum_{j=1...m} i_j$ and outputs $O = \sum_{j=1...n} o_j$ satisfy $I \geq O$

- If $I > O$ the difference $I - O$ is the transaction fee, paid to the miner who includes the transaction in a block

Bitcoin Addresses

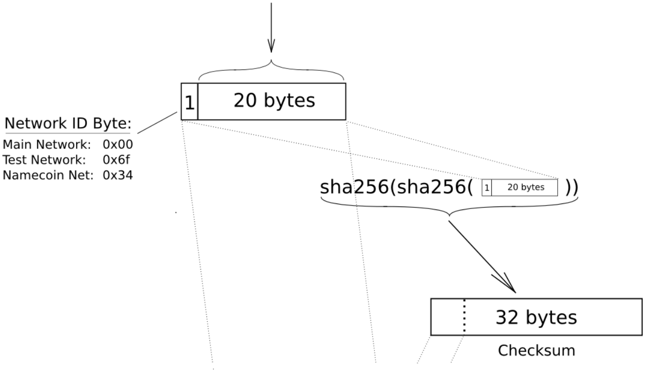
- An address is a destination where Bitcoin can be sent/held
- It is constructed from a user's ECDSA public key
- a 160-bit hash + checksum
- encoded using Base58 encoding

Public key to BTC address conversion



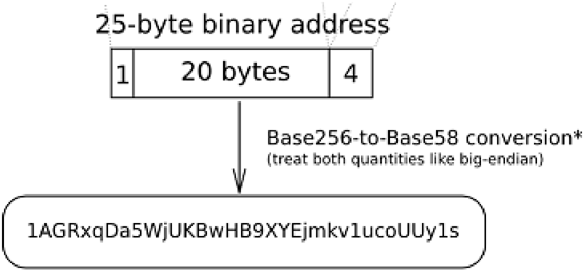
- Take either compressed (or uncompressed) SEC format
- SHA-256 the result and RIPEMD160 the result (aka HASH160)

Public key to BTC address conversion



- Prepend with network prefix (00 for mainnet, 6F for testnet)
- Add a 32-bit double-SHA256 checksum at the end

Public key to BTC address conversion



Transaction structure

- metadata
    - hash of the transaction (used as an identifier for the transaction)
    - version number
    - number of inputs
    - number of outputs
    - lock-time (earliest time the transaction can be included in a block)
    - size of the transaction
- inputs (an array)
- outputs (an array)

Transaction outputs

- Each output has the following structure
  - The value of the output (an amount of Bitcoin, in Satoshis)
  - A locking script that specifies a condition that needs to be met for the value to be released, and the output spent
- The simplest type of locking script describes a "recipient" of the output (holder of a private key)

Transaction outputs

- Each output has the following structure
  - The value of the output (an amount of Bitcoin, in Satoshis)
  - A locking script that specifies a condition that needs to be met for the value to be released, and the output spent
- The simplest type of locking script describes a “recipient” of the output (holder of a private key)
- Locking scripts are program written in Bitcoin Script, a small programming language

Transaction inputs

- Each input specifies the following
  - hash of a previous transaction
  - index of one of the outputs of the previous transaction
  - unlocking script size
  - unlocking script
    - proof that the unlocking condition of the previous output has been satisfied

Transaction verification

- In order for the transaction to be valid
  - the previous transaction output should be unspent
  - the unlocking script should satisfy the condition in the output's locking script

A simple transfer example

- Suppose <Transaction> includes a simple transfer of 100 Satoshis from Alice to Bob

- Alice output

A simple transfer example

- Suppose <Transaction> includes a simple transfer of 100 Satoshis from Alice to Bob

- Alice output
  - value: 100

A simple transfer example

- Suppose <Transaction> includes a simple transfer of 100 Satoshis from Alice to Bob

- Alice output
  - value: 100
  - locking script: "Check that the input consists of two values <sig> <PublicKey>, where
    1. hash(<PublicKey>) is <AliceAddress>, and
    2. <PublicKey> verifies that <sig> is a signature of hash(<Transaction>)"

A simple transfer example

- Suppose <Transaction> includes a simple transfer of 100 Satoshis from Alice to Bob

- Input
  - unlocking script: <hash(<Transaction>) signed AlicePrivateKey> <AlicePublicKey>
- Recall that <AliceAddress> = hash(<AlicePublicKey>)

Security Argument

- Note that anyone who knows <AliceAddress> is able to construct the output locking script
- This means that anyone could have sent the money to Alice by constructing a transaction with this output

Security Argument

- For the unlocking condition to evalue to True, we need:
  1. hash(<PublicKey>) = <AliceAddress>
     - The only value likely to satisfy this is <PublicKey> = <AlicePublicKey>
  2. <PublicKey>, i.e., <AlicePublicKey> verifies that <sig> is a signature of hash(<Transaction>)
     - The only person likely to have been able to construct such a <sig> is Alice, using <AlicePrivateKey>

Summary

- That is with this input/output pattern
  - To send someone money, you only need to know their address
  - To spend money sent to an address, you need to know the associated private key

Bitcoin Script

- Based on a stack-based memory model
- There are conditional statements, but no loops – this guarantees termination
- A program is a linear sequence of Opcodes (instructions) and data
- A data value is simply pushed onto the stack
- Each Opcode may do any of
    - consume some values from the top of the stack
    - calculate a value
    - push a result value onto the top of the stack

Bitcoin Script in more detail

An input I in a transaction T validly consumes an unspent output O, when executing

$$\textit{unlocking-script}(I);\ \textit{locking-script}(O)$$

(L to R) leaves the stack containing just value TRUE on termination

Opcode Examples

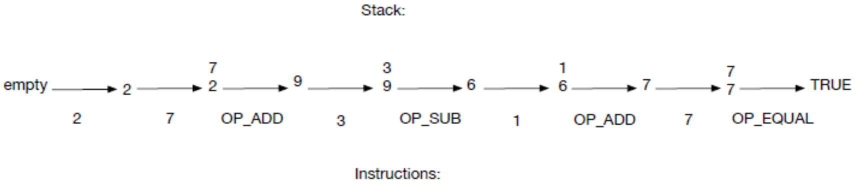
- OP_ADD
  - pop the two top values on the stack, add them, and push the result
- OP_SUB
  - pop the two top values on the stack, subtract first from second, and push the result
- OP_EQUAL
  - pop the two top values on the stack, test the values for equality, and push the Boolean result
- OP_DUP
  - duplicate the top value on the stack, i.e. push a copy onto the stack

Opcode Examples

- Example script

  2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL

- Executes as



Exercise

What does this return?

2 OP_DUP OP_ADD OP_DUP OP_ADD 4 OP_EQUAL

Some More (Cryptographic) Opcodes

- OP_HASH160
  - pop x, and push (RIPEMD(SHA256(x)))
- OP_HASH256
  - pop x, and push (SHA256(SHA256(x)))
- OP_CHECKSIG
  - pop K and S and push the result of checking if K is a public key that verifies S as a signature (by the corresponding private key) of the hash of the current transaction
- OP_EQUALVERIFY
  - first check equality of the top two values, if TRUE then run OP_CHECKSIG on the next two values
- Using these, we can write the "Simple Transfer" example in Script . . .

Pay to Public Key Hash Transaction Script

- Output locking script

  OP_DUP OP_HASH160 <recipient address> OP_EQUALVERIFY

- Input unlocking script

  <sig> <publicKey>

Pay to Public Key Hash Transaction Script