Dev++

ECC and Transactions

Connect to wifi
Install python3, virtualenv, git

```
$ git clone http://github.com/bitcoinedge/devplusplus
$ cd devplusplus
$ virtualenv -p python3 .venv
$ . .venv/bin/activate
$ pip install -r requirements.txt
$ jupyter notebook
```

Your web browser should open up a jupyter notebook

Class Structure

- Present some material
- Ask questions
- You have time to play with/study the code

What We’ll Cover

- Foundational Math
- Elliptic Curve Cryptography
- Transactions

Finite Fields

What Is a Finite Field?

- Set of numbers
- Finite
- Closed under +, -, *, /, except division by 0
- Used with Elliptic Curves for Cryptography
- Prime fields are the most interesting

Example

Prime Field of 19 (Denoted F_{19})

$F_{19} = \{0, 1, 2, \dots 18\}$

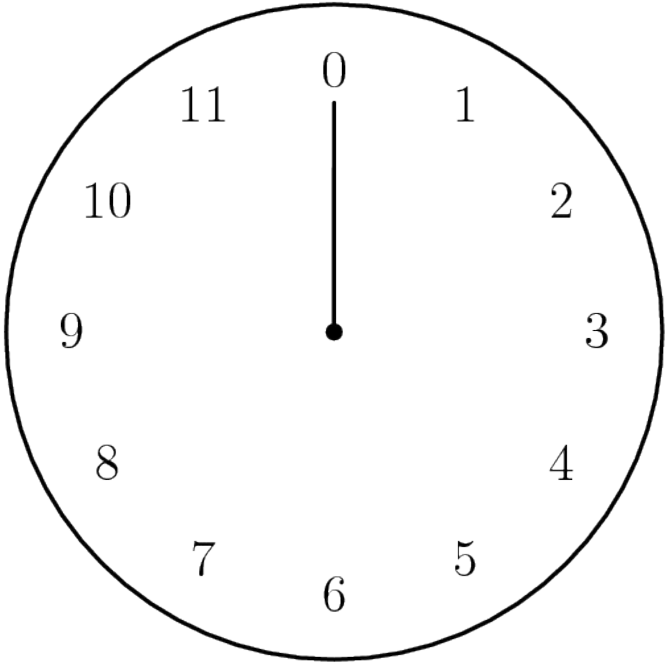
$F_{97} = \{0, 1, 2, \dots 96\}$

$F_{48947} = \{0, 1, 2, \dots 48946\}$

Modular Arithmetic

Remainder math

11 / 7 = 1 R 4, 38 / 12 = 3 R 2



Addition and Subtraction

Same as modulo P arithmetic (F_{19})

$11 + 6 = 17 \% 19 = 17$

$17 - 6 = 11 \% 19 = 11$

$8 + 14 = 22 \% 19 = 3$

$4 - 12 = -8 \% 19 = 11$

Multiplication and Exponentiation

Same as modulo P arithmetic (F_{19})

`2 * 4 = 8 % 19 = 8`

`7 * 3 = 21 % 19 = 2`

11^3 `= 1331 % 19 = 1`

Python: `pow(11, 3, 19) == 1`

Division

Inverse of Multiplication (F_{19})

$2 * 4 = 8 \Rightarrow 8 / 4 = 2$

$7 * 3 = 2 \Rightarrow 2 / 3 = 7$

$15 * 4 = 3 \Rightarrow 3 / 4 = 15$

$11 * 11 = 7 \Rightarrow 7 / 11 = 11$

Fermat's Little Theorem

$n^{p-1}=1 \mod p$

Works for all n if p is prime. This means that

$1/n=n^{-1}=n^{-1}*1=n^{-1}*n^{p-1}=n^{p-2} \mod p$

This is how we do division.

Division

So how do we calculate it? (F_{19})

$n^{p-1} = 1 \Rightarrow 1/n = n^{p-2}$

$2 / 3 = 2 * 1/3 = 2 * 3^{17} = 7$

$3 / 15 = 3 * 1/15 = 3 * 15^{17} = 4$

Python: `1/n = pow(n, p-2, p)`

Examples

```
from ecc import FieldElement

a = FieldElement(2, 19)
b = FieldElement(15, 19)

# Add
print(a+b) # 17
# Subtract
print(a-b) # 6
# Multiply
print(a*b) # 11
# Exponentiate
print(b**5) # 2
# Divide
print(a/b) # 9
```

Study

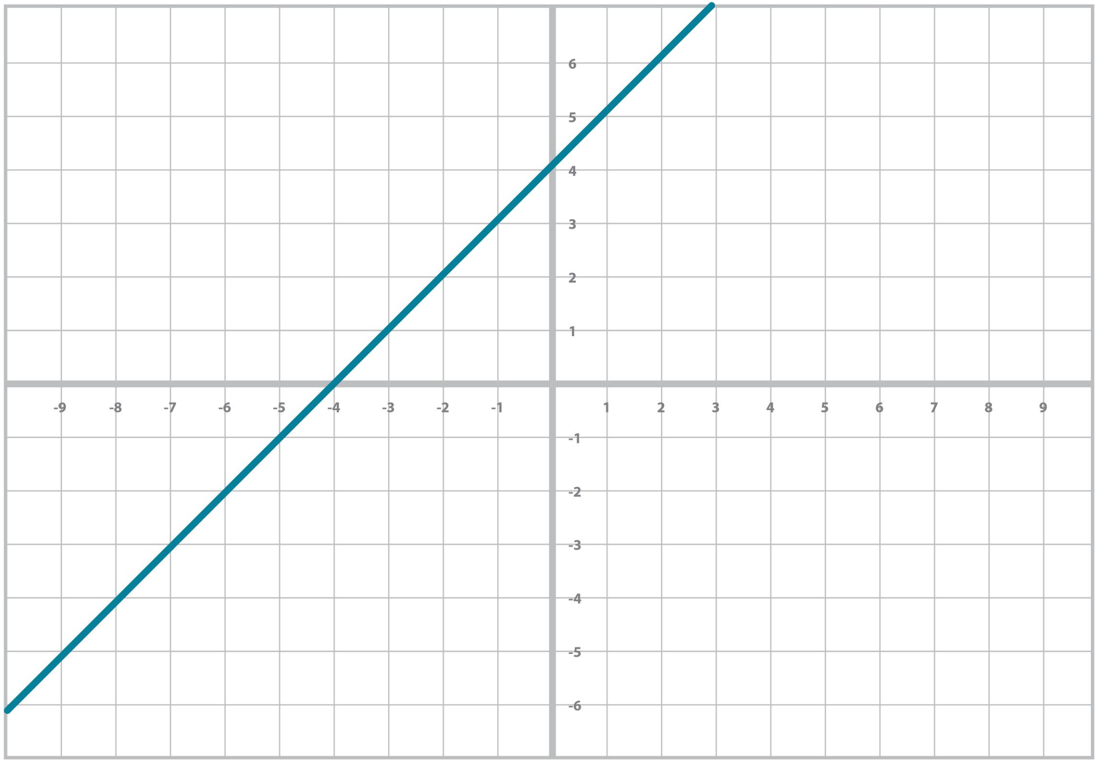
`ecc.py:FieldElement`

`ecc.py:FieldElementTest`

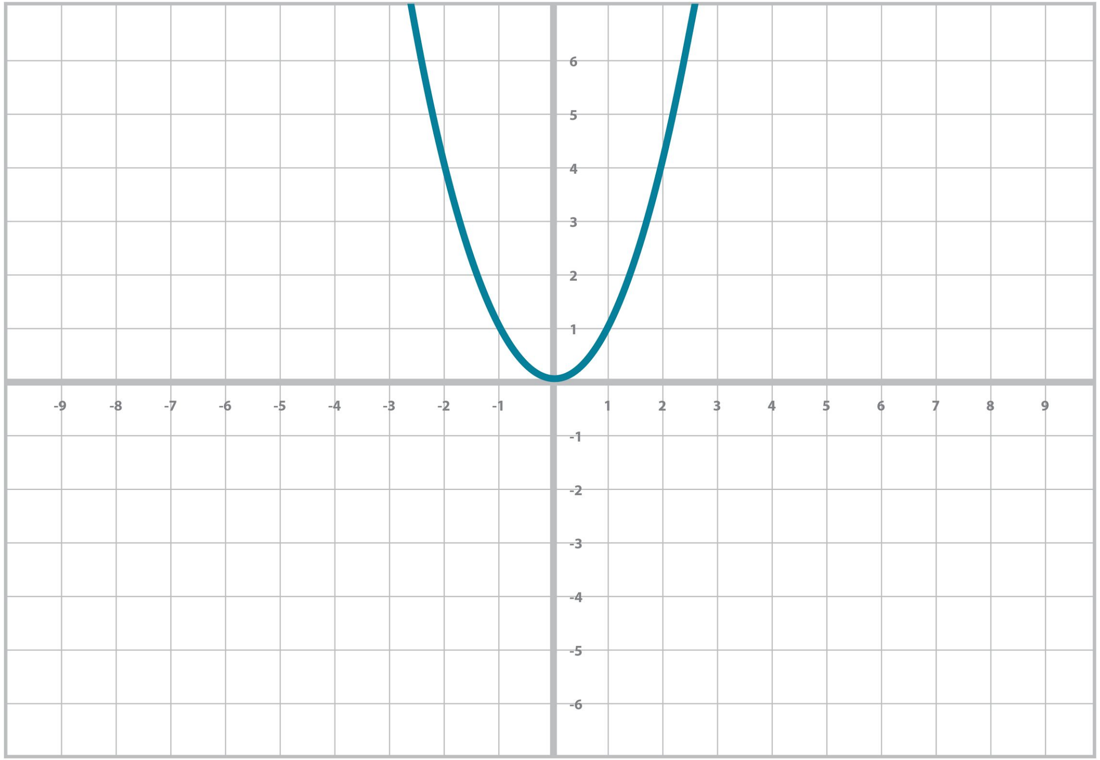
Questions?

Elliptic Curves

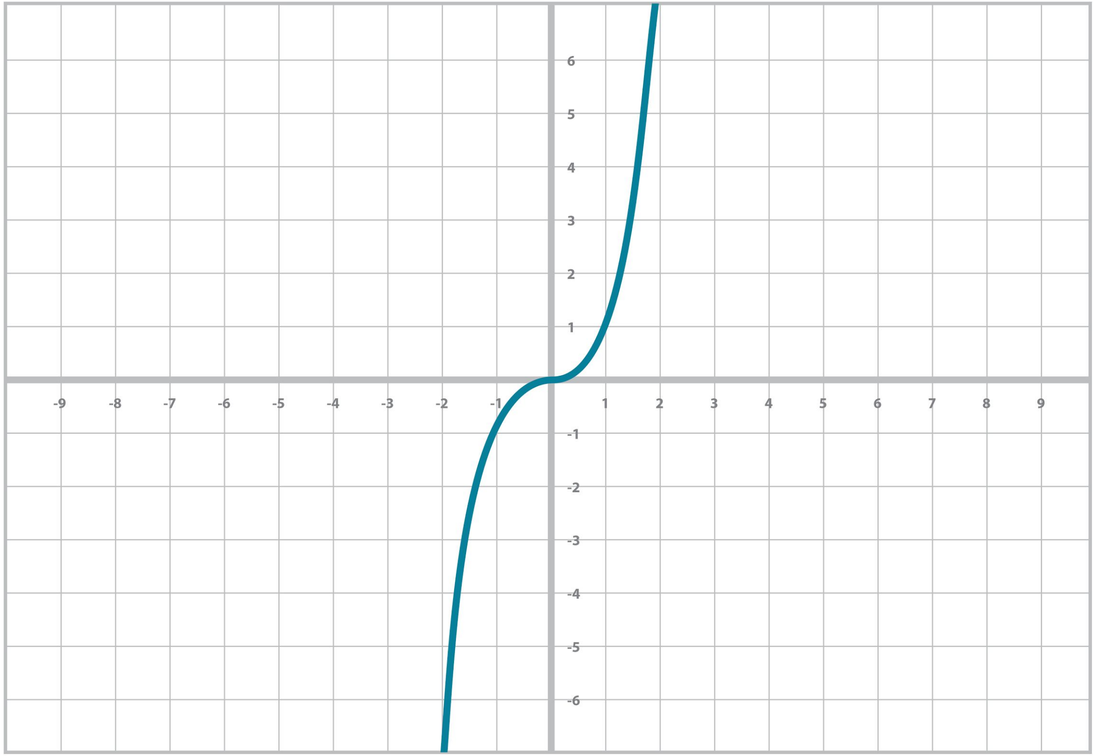
Linear (y = ax + b)



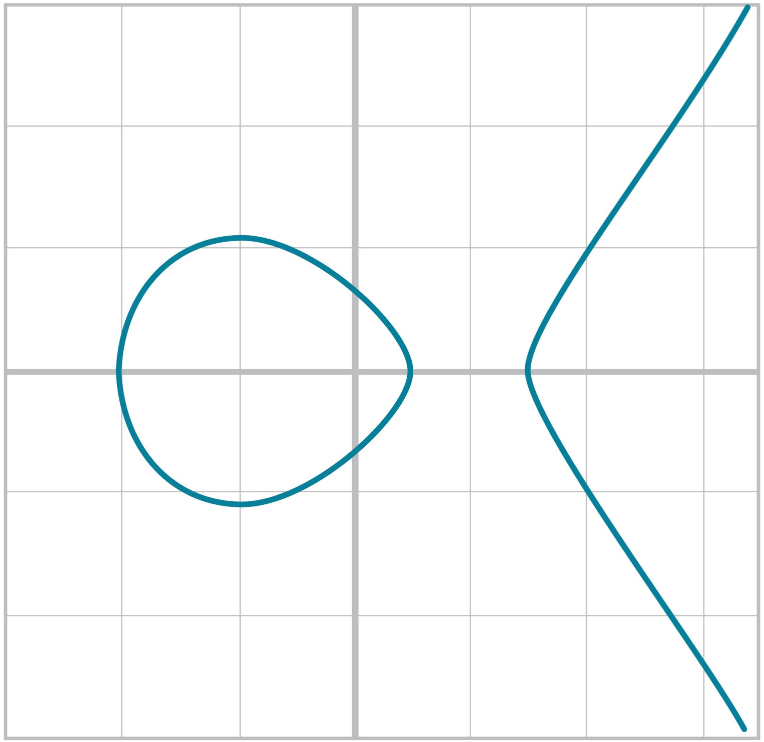
Quadratic: ($y = ax^2 + bx + c$)

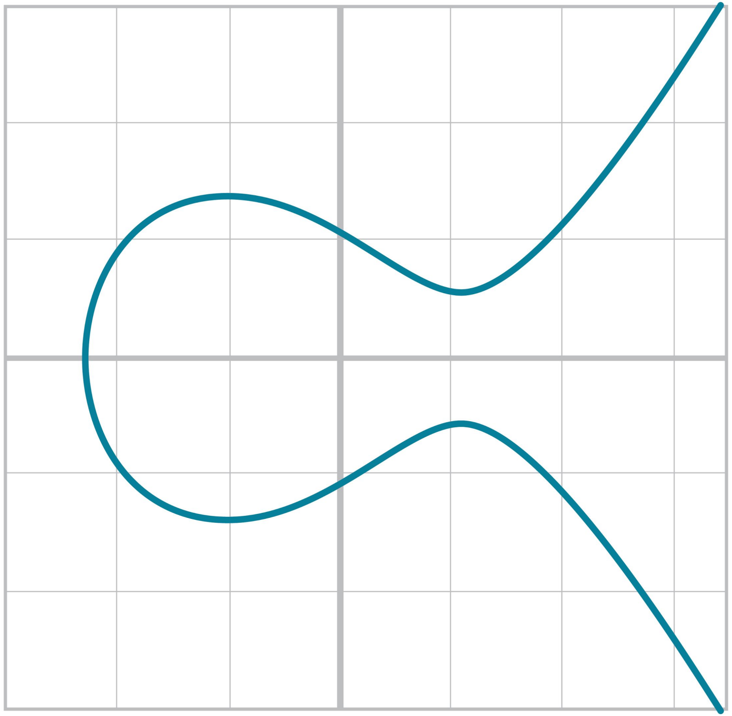


Cubic: ($y = ax^3 + bx^2 + cx + d$)

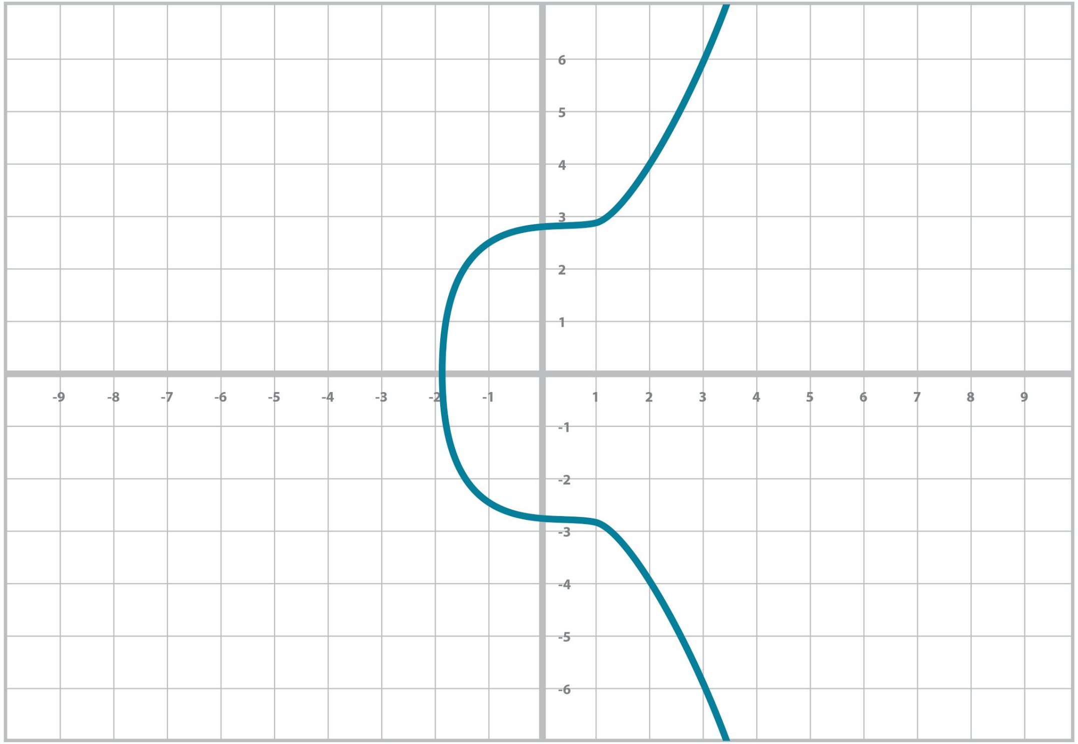


Elliptic: ($y^2 = x^3 + bx + c$)

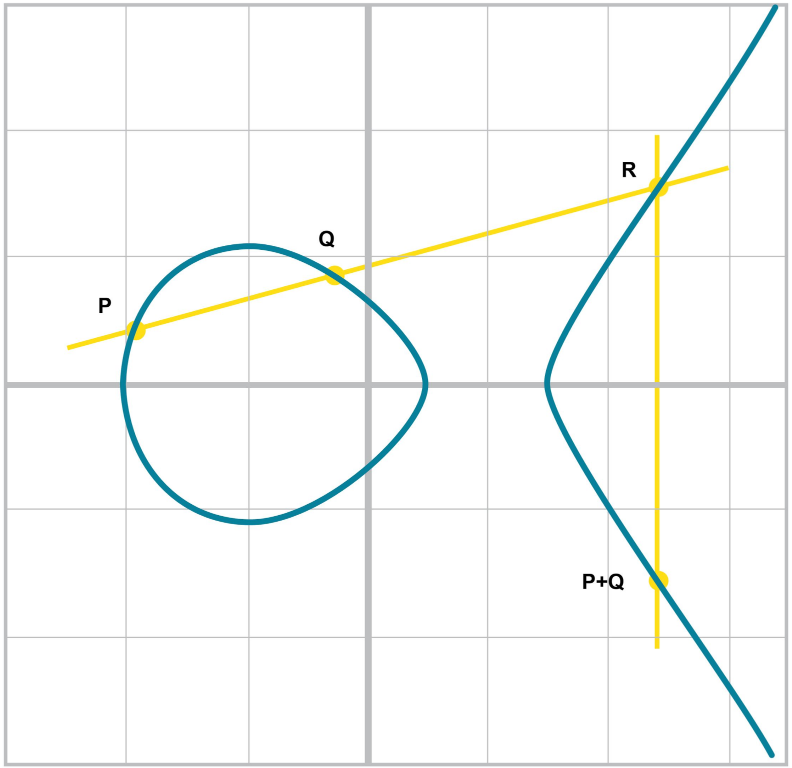




secp256k1: $y^2 = x^3 + 7$



Point Addition



Group Law for the point at ∞

$\texttt{Curve:}y^2=x^3+ax+b$

$(x_1,y_1)=(x_1,y_1)+(\infty,\infty)$

$(x_1,y_1)+(x_1,-y_1)=(\infty,\infty)$

Think zero

Group Law for $x_1 \neq x_2$

$$\mathrm{Curve:} y^2 = x^3 + ax + b$$

$$(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$$

$$s = (y_2 - y_1) / (x_2 - x_1)$$

$$x_3 = s^2 - x_1 - x_2$$

$$y_3 = s(x_1 - x_3) - y_1$$

Example

Curve: $y^2 = x^3 + 5x + 7$

What is (2,5) + (3,7)?

$s=(y_2-y_1)/(x_2-x_1)=(7-5)/(3-2)=2$

$x_3=s^2-x_1-x_2=2^2-2-3=-1$

$y_3=s(x_1-x_3)-y_1=2(2-(-1))-5=1$

(2,5) + (3,7) = (-1, 1)

Group Law for $x_1 = x_2 \quad y_1 = y_2$

Curve: $y^2 = x^3 + ax + b$

$(x_3, y_3) = (x_1, y_1) + (x_1, y_1)$

$s = (3x_1{}^2 + a) / (2y_1)$

$x_3 = s^2 - 2x_1$

$y_3 = s(x_1 - x_3) - y_1$

Examples

```
from ecc import Point

p0 = Point(x=None, y=None, a=5, b=7)
p1 = Point(x=-1, y=1, a=5, b=7)
p2 = Point(x=3, y=7, a=5, b=7)

# Add identity
print(p0+p1) # (-1,1)

# Add Different Points
print(p1+p2) # (0.25,-2.875)

# Add Same Points
print(p1+p1) # (18,-77)
```

Study

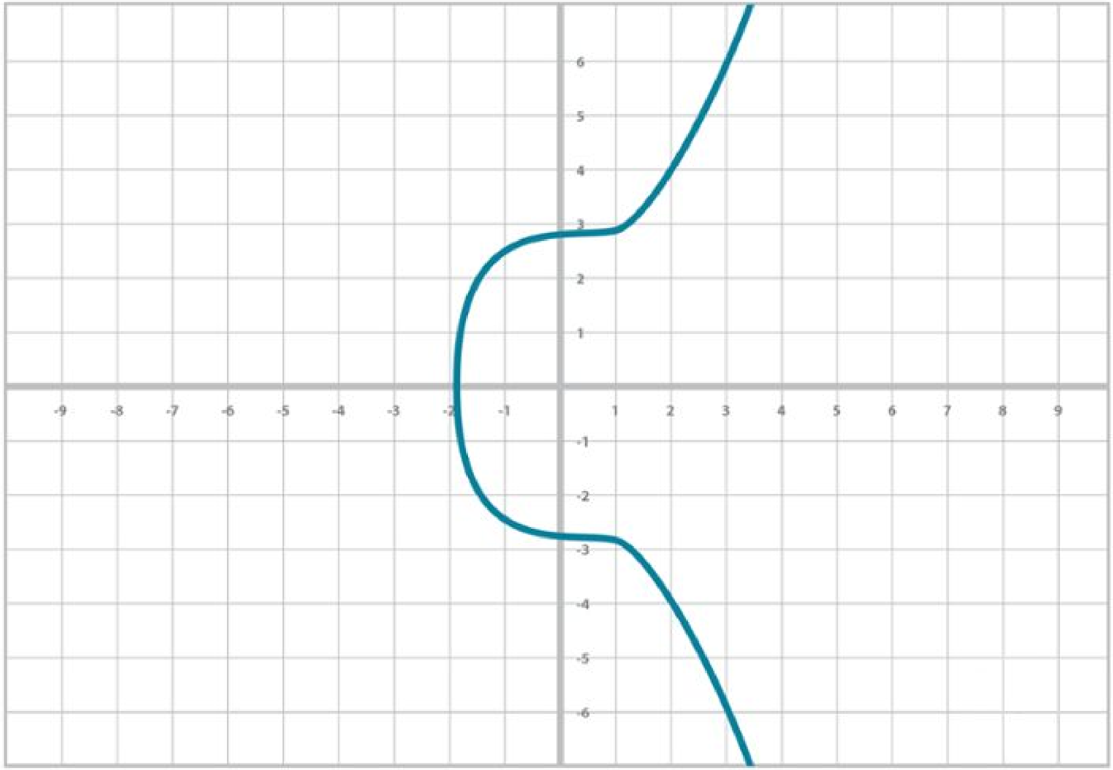
ecc.py:Point

ecc.py:PointTest

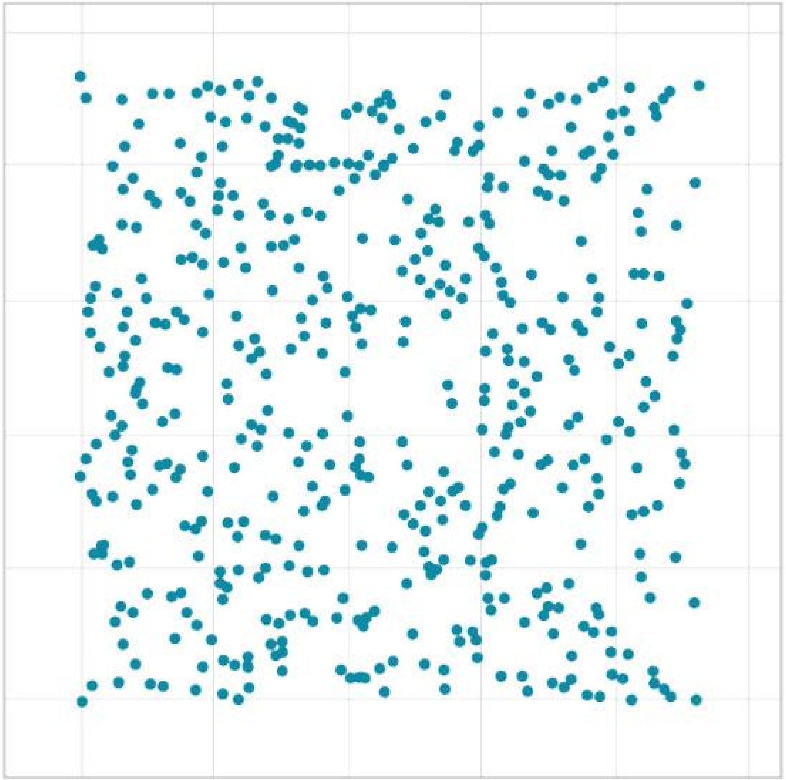
Questions?

Elliptic Curves over Finite Fields

Elliptic Curve over Reals



Elliptic Curve over Finite Field



Examples

```
from ecc import FieldElement, Point

a = FieldElement(0, 137)
b = FieldElement(7, 137)

p0 = Point(x=None, y=None, a=a, b=b)
p1 = Point(x=FieldElement(73, 137), y=FieldElement(128,
137), a=a, b=b)
p2 = Point(x=FieldElement(46, 137), y=FieldElement(22, 137),
a=a, b=b)

print(p1+p2)
print(p1+p1)
```

Study

```
ecc.py:ECCTest:test_on_curve
ecc.py:ECCTest:test_add1
```

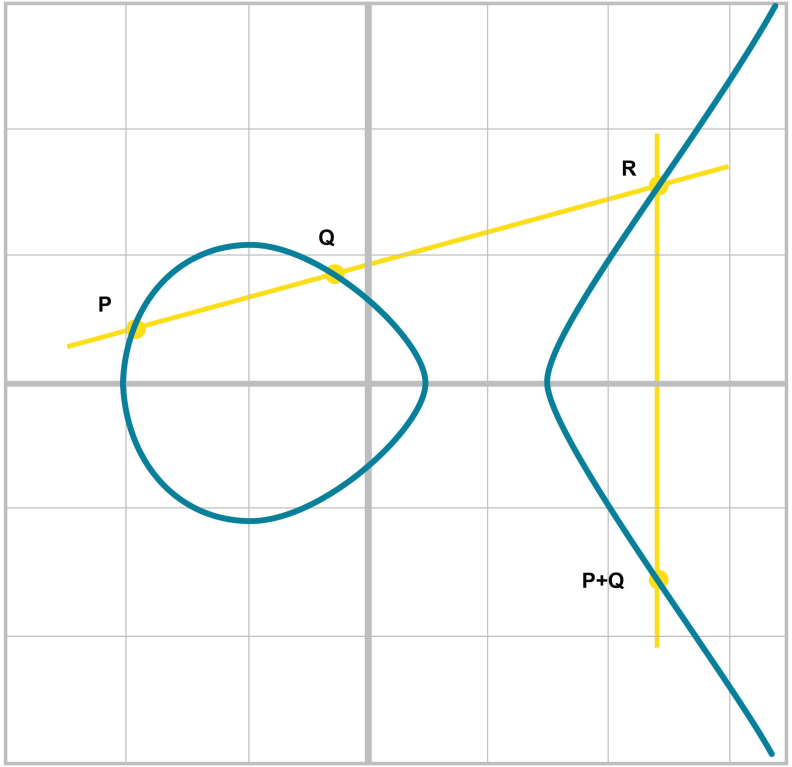
Questions?

Mathematical Group

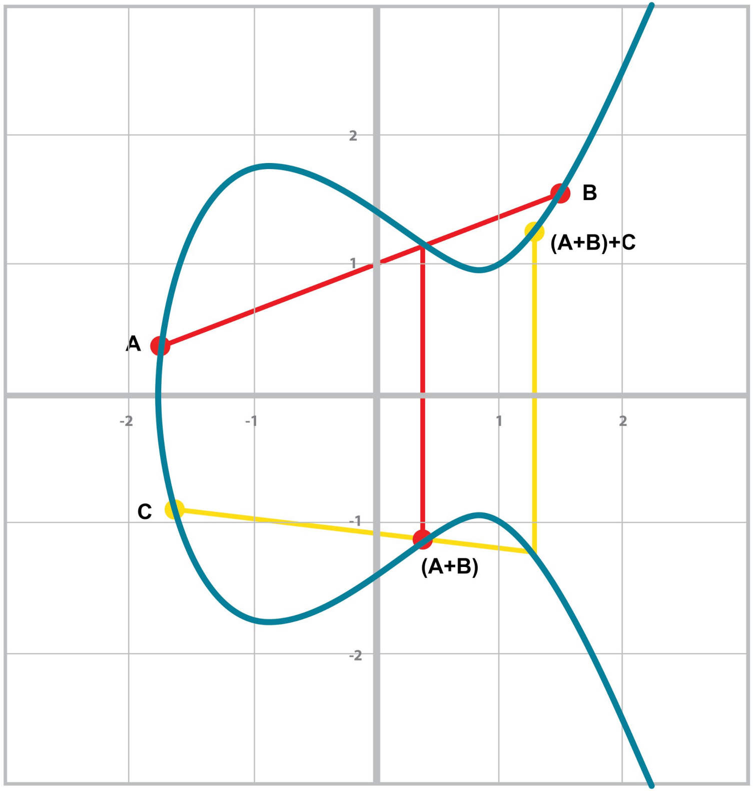
Mathematical Group

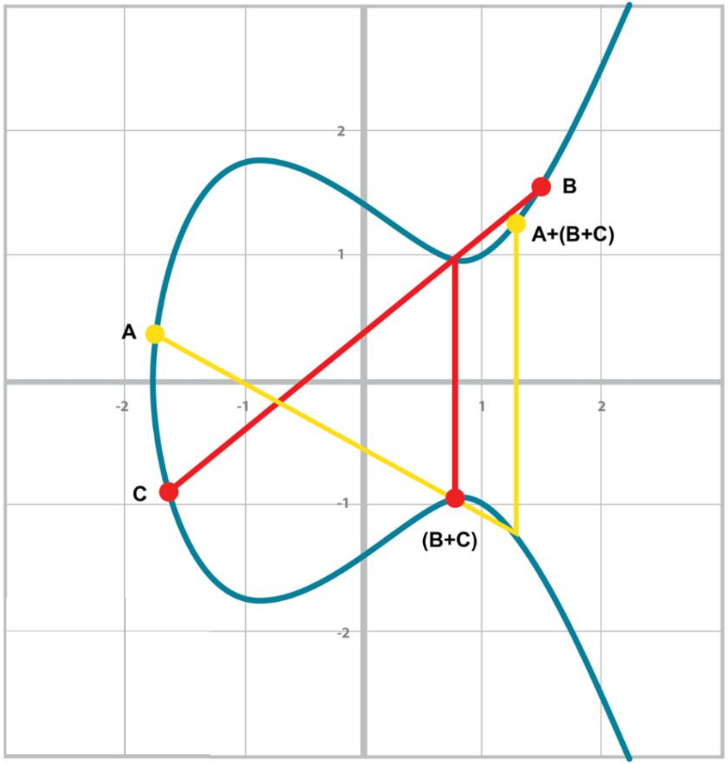
- Single operation
- Closed (if A, B in G, A+B in G)
- Associative ((A+B)+C = A+(B+C))
- Commutative (A+B=B+A)
- Invertible (if A in G, there’s a -A in G)
- Identity (0 exists)
- Point addition gets us a group!

Closed, Commutative, Invertible



Associative





Scalar Multiplication

Start with an Elliptic Curve over a Finite Field.

Pick a point G (generator point).

`G+G=2G, G+G+G=3G` … `nG` where `nG=0` (point at ∞)

`{0,G,2G,…(n-1)G}` is a finite group

Examples

```
from ecc import FieldElement, Point

a = FieldElement(0, 137)
b = FieldElement(7, 137)
p0 = Point(x=None, y=None, a=a, b=b)
p1 = Point(x=FieldElement(73, 137), y=FieldElement(128, 137), a=a, b=b)
current = p1
n = 1
while current != p0:
    current += p1
    n += 1
print(n, p1, n*p1) # order of p1 is 69
```

Study

`ecc.py:ECCTest:test_rmul`

Questions?

Scalar Multiplication

- Imagine a really large group $n \sim 2^{256}$
- $P=sG$ where s is really, really large
- Finding P when we know s is easy
- Finding s when we know P is not
- Sometimes referred to as “Secret Exponent”
- $P=G^s$ => $\mathrm{Log}_G P=s$ (Discrete Log Problem)
- Convention: lower-case letters for scalar, upper-case letters for points

Defining an Elliptic Curve

- Elliptic Curve Equation (a and b of $y^2=x^3+ax+b$)
- Finite Field Prime Number (p)
- Generator Point (G)
- Number of points in the group (n)

secp256k1

- Equation $y^2=x^3+7$ (a=0, b=7)
- Prime Field (p) = $2^{256}-2^{32}-977$
- Generator Point (G) =
  (79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798,
  483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8)
- Order (n) =
  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
- SEC = Standards for Efficient Cryptography
- 256 = number of bits in the prime field

2^{256} is really big

- $2^{256} \sim 10^{77}$
- Number of atoms in and on earth ~ 10^{50}
- Number of atoms in the solar system ~ 10^{57}
- Number of atoms in the galaxy ~ 10^{68}
- Number of atoms in the universe ~ 10^{80}
- Trillion computers doing a trillion operations every picosecond (10^{-12} seconds) for a trillion years < 10^{56} operations.

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

Getting a public key from private

```
from ecc import G

secret = 999
point = secret*G
print(point)
Point(9680241112d370b56da22eb535745d9e314380e568229e09f72410
66003bc471,
ddac2d377f03c201ffa0419d6596d10327d6c70313bb492ff495f946285d
8f38)
```

Study

```
ecc.py:S256Field
ecc.py:S256Point
```

Questions?

Bitcoin Addresses

SEC Format

- Public Key (point on curve) serialized
- Uncompressed

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

Addresses

- Take either compressed or uncompressed SEC format
- SHA-256 the result and then RIPEMD160 the result (aka HASH160)
- Prepend with network prefix (00 for mainnet, 6F for testnet)
- Add a 32-bit double-SHA256 checksum at the end
- Encode in Base58

Example

```
from ecc import G

secret = 999
point = secret*G
print(point.address(compressed=True, testnet=False))
print(point.address(compressed=False, testnet=False))
print(point.address(compressed=True, testnet=True))
print(point.address(compressed=False, testnet=True))
```

Study

`ecc.py:S256Test`

Questions?

ECDSA

Elliptical Curve Digital Signature Algorithm

Intuition

- $sG=P$
- $uG+vP$ where $u,v\neq 0$
- Say you can choose u and v
- Can only manipulate the sum if you know how G and P are related (that is, you know the private key)
- $uG+vP=uG+vsG=(u+sv)G$
- If you know s, you can manipulate u+sv, if you don't you can't.

Signature Algorithm

- Start with the hash of what you're signing (z)
- Next assume your secret is e and the public point `P=eG`
- Get a new random number k
- Compute `kG.` The x coordinate = r
- Compute `s=(z+re)/k` (Division is the same as field division: `1/x = pow(x,n-2,n)`)
- Signer can compute s since he has e, nobody else can compute s.
- Signature is simply the pair, (r, s)
- Note s has to be less than n/2. If s>n/2, use n-s instead.

Verification Algorithm

- Start with the hash of what you're 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`
- `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 the x coordinate matches r, you have a valid sig.

Example

```
from ecc import PrivateKey

z = 432089432098342098234089098423098324089
secret = 999
priv_key = PrivateKey(secret)
pub_key = priv_key.point
sig = priv_key.sign(z)
print(sig)
print(pub_key.verify(z, sig)) # True
```

Study

```
ecc.py:Signature
ecc.py:SignatureTest
ecc.py:PrivateKey
ecc.py:PrivateKeyTest
```

Questions?

DER Signature Format

Encoding r and s

DER Format

Signature (r,s) serialized

`3045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf213 20b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801 c31967743a9c8e10615bed`

- `30 - Marker`
- `45 - Length of sig`
- `02 - Marker for r value`
- `21 - r value length`
- `00ed…8f - r value`
- `02 - Marker for s value`
- `20 - s value length`
- `7a98...ed - s value`

Example

```
from binascii import hexlify
from random import randint
from ecc import PrivateKey

z = randint(0, 2**256)
secret = 999
priv_key = PrivateKey(secret)
sig = priv_key.sign(z)
print(hexlify(sig.der()))
```

Study

```
ecc.py:Signature:serialize
ecc.py:Signature:test_der
```

Questions?

Transaction Structure

What is a Transaction?

- Assignment of bitcoin from one script to another.
- Note addresses are really compressed scripts.

Bitcoin Transaction Elements

- Version (4 bytes)
- Inputs
- Outputs
- Locktime

Inputs

- Two types of inputs
  - Coinbase
  - Previous transaction output (a.k.a. tx out, utxo, spendable, outpoint)

Input

- Elements
  - Previous tx hash
  - Vout (output index in that transaction)
  - Sequence - Used for RBF
  - Scriptsig - involves SCRIPT language
- Note: no amount here! You have to look it up.

- All nodes have to validate these inputs as legitimate.

Output

- Elements
  - Amount
  - ScriptPubKey - involves SCRIPT language
- What we think of as assigning to an address is actually a script in ScriptPubKey
- Note: Amount can be zero in certain instances (OP_RETURN)

Locktime

- Designed to tell nodes not to let a tx into a block until a certain time or a certain block height

Bitcoin Transaction

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7
d1000000006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf2
1320b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a
9c8e10615bed01210349fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213
bf016b278afeffffff02a135ef01000000001976a914bc3b654dca7e56b04dca18f2566c
daf02e8d9ada88ac99c39800000000001976a9141c4bc762dd5423e332166702cb75f40d
f79fea1288ac19430600
```

`01000000` - version

`01` - # of inputs

`813f...d1` - previous tx hash

`00000000` - previous tx index

`6b00...8a` - scriptSig

`feffffff` - sequence

`02` - # of outputs

`a135...00` - output amounts

`1976...ac` - scriptPubKey

`19430600` - locktime

Example

```
from binascii import unhexlify
from io import BytesIO
from tx import Tx

raw_tx =
BytesIO(unhexlify('0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71b
f8303c6a989c7d1000000006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c503
1ccfcf21320b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743
a9c8e10615bed01210349fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016
b278afeffffff02a135ef01000000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ad
a88ac99c39800000000001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430
600'))
tx_obj = Tx.parse(raw_tx)
print(tx_obj)
```

Study

`tx.py:Tx`

`tx.py:TxTest`

Questions?

Script

What is SCRIPT?

- Limited Programming Language
- Not Turing Complete.
- Programmable way to assign bitcoins
- Addresses are compressed scripts

How does SCRIPT work?

- There are two types of items: elements and operations.
- Elements are just data (signatures, keys, hashes, etc)
- Operations do something to elements.
- Each element is added to a stack, operations do something to the stack.
- At the end of processing all the items, there must be a single element that’s not zero left on the stack to evaluate to True.

Some common operations

- OP_DUP - duplicates the top element in the stack and puts it on top
- OP_HASH160 - Does a SHA256 and then a RIPEMD160 to the top element.
- OP_CHECKSIG - Takes the top two elements, the first being the pubkey, the second being the signature and checks if the signature is valid for the current transaction.
- OP_RETURN - Marks transaction as invalid, but also allows 80 bytes of data to be put in.

Parsing SCRIPT

- Each byte is interpreted as an integer.
- If byte is between 1 and 75 inclusive, the next n bytes are an element.
- Otherwise, byte is an operation based on a lookup table

```
0x00 - OP_0, Put a 0 on top of the stack
0x05 - Next 5 bytes are an element
0x48 - Next 72 bytes are an element
0x76 - OP_DUP, Put a copy of the top element of the stack on
top of the stack
0x93 - OP_ADD, Take the top two elements, add them and put
on top of the stack
0xa9 - OP_HASH160, Perform a HASH160 to the top element of
the stack
… Many more
```

Some common elements

- Public keys - SEC Format (33 or 65 bytes)
- Signatures - DER Format (71, 72 or 73 bytes)
- Hash160 - 20 bytes

Common Scripts

- Addresses are compressed scripts
- p2pk - pay to pub key
- p2pkh - pay to pub key hash
- p2sh - pay to script hash
- p2wpk - pay to witness pub key
- p2wsh - pay to witness script hash

SCRIPT Validation

- The ScriptSig field of the input has SCRIPT items which are processed one at a time until there are no more items
- Every non-coinbase input must specify the previous transaction and index.
- This outpoint has a ScriptPubKey, these SCRIPT items are processed one at a time until there are no more items
- If the result of the processing leaves a non-zero element, the SCRIPT is considered valid, otherwise, the SCRIPT is invalid.

P2PK - First Standard SCRIPT

Pay-to-Pubkey

scriptPubKey (receiving)

410411db93e1dcdb8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5cb2e0eaddfb84ccf9744464f82e160bfa9b8b64f9d4c03f999b8643f656b412a3ac

41 - length of the pubkey
0411...a3 - <pubkey>
ac - OP_CHECKSIG

scriptSig (spending)

47304402204e45e16932b8af514961a1d3a1a25fdf3f4f7732e9d624c6c61548ab5fb8cd410220181522ec8eca07de4860a4acdd12909d831cc56cbbac4622082221a8768d1d0901

47 - length of the signature
3044...01 - <signature>

P2PK - First Standard SCRIPT

scriptPubKey	scriptSig	Script
<pubkey>	<signature>	<signature>
OP_CHECKSIG		<pubkey>
		OP_CHECKSIG

P2PK

Script

Stack

Processing

<signature>

<pubkey>

OP_CHECKSIG

P2PK

Script	Stack	Processing
<pubkey>		
OP_CHECKSIG	<signature>	

P2PK

Script

Stack

Processing

<pubkey>

OP_CHECKSIG

<signature>

P2PK

Script

Stack

<pubkey>

<signature>

Processing

OP_CHECKSIG

`OP_CHECKSIG` - Checks if the signature is valid for the current transaction. Puts 1 back if valid, 0 otherwise.

P2PK - First Standard SCRIPT

Script	Stack	Processing
	1	

OP_CHECKSIG - Checks if the signature is valid for the current transaction. Puts 1 back if valid, 0 otherwise.

P2PKH - Shorter & more secure

- These are the addresses that start with a “1”
- Shorter due to use of RIPEMD160
- More Secure due to requiring both ECC Discrete Log and Hash pre-images being needed.

P2PKH - Shorter & more secure

Pay-to-Pubkey-Hash

scriptPubKey (receiving)

```
76a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac

76 - OP_DUP
a9 - OP_HASH160
14 - Length of <hash>
bc3b...da - <hash>
88 - OP_EQUALVERIFY
ac - OP_CHECKSIG
```

P2PKH - Shorter & more secure

Pay-to-Pubkey-Hash

scriptSig (spending)

483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf2
1320b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f678
01c31967743a9c8e10615bed01210349fc4e631e3624a545de3f89f5d868
4c7b8138bd94bdd531d2e213bf016b278a

48 - Length of <signature>
30...01 - <signature>
21 - Length of <pubkey>
0349...8a - <pubkey>

P2PKH

scriptPubKey

```
OP_DUP
OP_HASH160
<hash>
OP_EQUALVERIFY
OP_CHECKSIG
```

scriptSig

<signature>

<pubkey>

Script

<signature>

<pubkey>

```
OP_DUP
OP_HASH160
<hash>
OP_EQUALVERIFY
OP_CHECKSIG
```

P2PKH

Script

Stack

Processing

<signature>

<pubkey>

OP_DUP

OP_HASH160

<hash>

OP_EQUALVERIFY

OP_CHECKSIG

P2PKH

Script

<pubkey>

OP_DUP

OP_HASH160

<hash>

OP_EQUALVERIFY

OP_CHECKSIG

Stack

<signature>

Processing

P2PKH

Script

Stack

Processing

OP_DUP

OP_HASH160

<hash>

OP_EQUALVERIFY

OP_CHECKSIG

<pubkey>

<signature>

P2PKH

Script

OP_HASH160

<hash>

OP_EQUALVERIFY

OP_CHECKSIG

Stack

<pubkey>

<signature>

Processing

OP_DUP

OP_DUP - duplicates the top element in the stack and puts it on top

P2PKH

Script

Stack

Processing

Script	Stack
OP_HASH160	
<hash>	<pubkey>
OP_EQUALVERIFY	<pubkey>
OP_CHECKSIG	<signature>

`OP_DUP` - duplicates the top element in the stack and puts it on top

P2PKH

Script	Stack	Processing
<hash>	<pubkey>	
OP_EQUALVERIFY	<pubkey>	
OP_CHECKSIG	<signature>	OP_HASH160

OP_HASH160 - Does a SHA256 and then a RIPEMD160 to the top element.

P2PKH

Script	Stack	Processing
<hash>	<hash>	
OP_EQUALVERIFY	<pubkey>	
OP_CHECKSIG	<signature>	

`OP_HASH160` - Does a SHA256 and then a RIPEMD160 to the top element.

P2PKH

Script

Stack

Processing

OP_EQUALVERIFY

OP_CHECKSIG

<hash>

<hash>

<pubkey>

<signature>

P2PKH

Script

Stack

Processing

<hash>

<hash>

<pubkey>

OP_CHECKSIG

<signature>

OP_EQUALVERIFY

OP_EQUALVERIFY - Checks that the top two elements are equal. If not, fails the whole script.

P2PKH

Script

Stack

Processing

`OP_EQUALVERIFY` - Checks that the top two elements are equal. If not, fails the whole script.

<pubkey>

<signature>

`OP_CHECKSIG`

P2PKH

Script | Stack | Processing

<pubkey>

<signature>

OP_CHECKSIG

OP_CHECKSIG - Checks if the signature is valid for the current transaction. Puts 1 back if valid, 0 otherwise.

P2PKH

Script | Stack | Processing

Script	Stack	Processing
	1	

`OP_CHECKSIG` - Checks if the signature is valid for the current transaction. Puts 1 back if valid, 0 otherwise.

Study

```
script.py:Script
script.py:ScriptTest
```

Questions?

Transaction Validation

Validation

0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf
8303c6a989c7d1000000006b483045022100ed81ff192e75a3fd2304004d
cadb746fa5e24c5031ccfcf21320b0277457c98f02207a986d955c6e0cb3
5d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e
631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afe
ffffff02a135ef01000000001976a914bc3b654dca7e56b04dca18f2566c
daf02e8d9ada88ac99c39800000000001976a9141c4bc762dd5423e33216
6702cb75f40df79fea1288ac19430600

Check the inputs that they're unspent

[d1c789a9c60383bf715f3f6ad9d14b91fe55f3deb369fe5d9280cb1a0179
3f81]()

Note you need the entire blockchain to check this.

Validation

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf
8303c6a989c7d1000000006b483045022100ed81ff192e75a3fd2304004d
cadb746fa5e24c5031ccfcf21320b0277457c98f02207a986d955c6e0cb3
5d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e
631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afe
ffffff02a135ef01000000001976a914bc3b654dca7e56b04dca18f2566c
daf02e8d9ada88ac99c39800000000001976a9141c4bc762dd5423e33216
6702cb75f40df79fea1288ac19430600
```

Check the amounts and make sure that the inputs >= outputs.

Inputs = 42505594

Outputs = 32454049 + 10011545 = 42465594

42505594 >= 42465594 (note difference is the tx fee, which goes to the miner)

Tx Fee = 42505594 - 42465594 = 40000

Validation

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf
8303c6a989c7d1000000006b483045022100ed81ff192e75a3fd2304004d
cadb746fa5e24c5031ccfcf21320b0277457c98f02207a986d955c6e0cb3
5d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e
631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afe
ffffff02a135ef01000000001976a914bc3b654dca7e56b04dca18f2566c
daf02e8d9ada88ac99c39800000000001976a9141c4bc762dd5423e33216
6702cb75f40df79fea1288ac19430600
```

Check the scriptSigs for inputs are valid (that is, combined script evals to TRUE).

We'll cover that part (SCRIPT) later.

In practice, this means checking that the signature in the scriptSig is valid.

The best instructions for validating signatures are found here.

Validation

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf
8303c6a989c7d10000000000feffffff02a135ef0100000000001976a914bc
3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a914bc
76a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

To check the sig, we have to substitute the scriptSig with 00 or, an empty scriptSig, for every input.

Validation

`0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf`
`8303c6a989c7d1000000001976a914a802fc56c704ce87c42d7c92eb75e7`
`896bdc41ae88acfeffffff02a135ef01000000001976a914bc3b654dca7e`
`56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4b`
`c762dd5423e332166702cb75f40df79fea1288ac19430600`

We substitute the scriptSig of the input we're signing with the scriptPubKey from the transaction output that we're spending.

Look up the tx output from the blockchain:

[d1c789a9c60383bf715f3f6ad9d14b91fe55f3deb369fe5d9280cb1a01793f81]

Reveals this scriptPubKey:

`76a914a802fc56c704ce87c42d7c92eb75e7896bdc41ae88ac`

We prepend with the length (`19`) to complete it.

Validation

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf
8303c6a989c7d1000000001976a914a802fc56c704ce87c42d7c92eb75e7
896bdc41ae88acfeffffff02a135ef01000000001976a914bc3b654dca7e
56b04dca18f2566cdaf02e8d9ada88ac99c398000000000001976a9141c4b
c762dd5423e332166702cb75f40df79fea1288ac1943060001000000
```

To check the sig, we also have to append the “hash type” or what the signature is good for. In this case, we’re appending “SIGHASH_ALL” or “This sig is only good if the entire transaction goes through”. There are others have different restrictions, but SIGHASH_ALL is the default and the most widely used.

Bitcoin Cash uses SIGHASH_ALL & SIGHASH_FORKID here for replay protection.

Validation

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf
8303c6a989c7d1000000001976a914a802fc56c704ce87c42d7c92eb75e7
896bdc41ae88acfeffffff02a135ef01000000001976a914bc3b654dca7e
56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4b
c762dd5423e332166702cb75f40df79fea1288ac1943060001000000
```

Double-sha256 this to get the hash that’s being signed (z)

Validation

0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf
8303c6a989c7d1000000006b483045022100ed81ff192e75a3fd2304004d
cadb746fa5e24c5031ccfcf21320b0277457c98f02207a986d955c6e0cb3
5d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e
631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afe
ffffff02a135ef01000000001976a914bc3b654dca7e56b04dca18f2566c
daf02e8d9ada88ac99c39800000000001976a9141c4bc762dd5423e33216
6702cb75f40df79fea1288ac19430600

We can get the signature from the scriptSig

Validation

0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf
8303c6a989c7d1000000006b483045022100ed81ff192e75a3fd2304004d
cadb746fa5e24c5031ccfcf21320b0277457c98f02207a986d955c6e0cb3
5d446a89d3f56100f4d7f67801c31967743a9c8e10615bed0121**0349fc4e**
631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afe
ffffff02a135ef01000000001976a914bc3b654dca7e56b04dca18f2566c
daf02e8d9ada88ac99c39800000000001976a9141c4bc762dd5423e33216
6702cb75f40df79fea1288ac19430600

We can also get the public key from the scriptSig

Example

```
from binascii import unhexlify
from io import BytesIO
from tx import Tx

raw_tx =
BytesIO(unhexlify('0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71b
f8303c6a989c7d1000000006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c503
1ccfcf21320b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743
a9c8e10615bed01210349fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016
b278afeffffff02a135ef01000000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ad
a88ac99c39800000000001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430
600'))
tx_obj = Tx.parse(raw_tx)
for i, tx_in in enumerate(tx_obj.tx_ins):
    print(tx_obj.verify_input(i))
```

Study

`tx.py:Tx:verify_input`

`tx.py:TxTest:test_verify_input1`

Questions?

Pay to Script Hash

Bare Multisig

scriptPubKey (receiving) - 1 of 2

```
514104fcf07bb1222f7925f2b7cc15183a40443c578e62ea17100aa3b44b
a66905c95d4980aec4cd2f6eb426d1b1ec45d76724f26901099416b9265b
76ba67c8b0b73d210202be80a0ca69c0e000b97d507f45b98c49f58fec66
50b64ff70e6ffccc3e6d0052ae
```

```
51 - OP_1
41 - Length of <pubkey1>
40fc...3d - <pubkey1>
21 - Length of <pubkey2>
0202...00 - <pubkey2>
52 - OP_2
ae - OP_CHECKMULTISIG
```

Bare Multisig

scriptSig (spending) - 1 of 2

```
00483045022100e222a0a6816475d85ad28fbeb66e97c931081076dc9655
da3afc6c1d81b43f9802204681f9ea9d52a31c9c47cf78b71410ecae6188
d7c31495f5f1adfe0df5864a7401
```

```
00 - OP_0
48 - Length of <signature1>
3045...01 - <signature1>
```

Bare Multisig

scriptPubKey

m
<pubkey1>
<pubkey2>
<pubkey...>
<pubkeyn>
n
OP_CHECKMULTISIG

scriptSig

x
<signature1>
<signature2>
<signature...>
<signaturem>

Script

x
<signature1>
<signature...>
<signaturem>
m
<pubkey1>
<pubkey...>
<pubkeyn>
n
OP_CHECKMULTISIG

Bare Multisig

Script

Stack

Processing

- x
- <signature1>
- <signature...>
- <signaturem>
- m
- <pubkey1>
- <pubkey...>
- <pubkeyn>
- n
- OP_CHECKMULTISIG

Bare Multisig

Script	Stack	Processing
<signature1>		
<signature...>		
<signaturem>		
m		
<pubkey1>		
<pubkey...>		
<pubkeyn>		
n		
OP_CHECKMULTISIG	x	

Bare Multisig

Script

Stack

Processing

m

<pubkey1>

<pubkey...>

<pubkeyn>

n

OP_CHECKMULTISIG

<signaturem>

<signature...>

<signature1>

x

Bare Multisig

Script	Stack	Processing
	n	
	<pubkeyn>	
	<pubkey...>	
	<pubkey1>	
	m	
	<signaturem>	
	<signature...>	
	<signature1>	
OP_CHECKMULTISIG	x	

Bare Multisig

Script

Stack

n

<pubkeyn>

<pubkey...>

<pubkey1>

m

<signaturem>

<signature...>

<signature1>

x

Processing

OP_CHECKMULTISIG

`OP_CHECKMULTISIG` - Checks if m of the signatures are valid of the n public keys for current transaction. Puts 1 back if valid, 0 otherwise.

Bare Multisig

Script

Stack

Processing

`OP_CHECKMULTISIG` - Checks if m of the signatures are valid of the n public keys for current transaction. Puts 1 back if valid, 0 otherwise.

1

Bare Multisig

- The `x` can be anything. It's required because of a bug in `OP_CHECKMULTISIG` and would require a hard fork to fix.
- There's no way to make this an address. It's too long.
- Big Transaction Output for the UTXO set
- This was abused.

Bare Multisig

How is the whitepaper decoded from the blockchain (Tx with ~1000x m of n multisig outputs)

▲

16

▼

★

3

The whitepaper is apparently encoded at 54e48e5f5c656b26c3bca14a8c95aa583d07ebe84dde3b7dd4a78f4e4186e713, which is an *m of n multisig* Tx with **947 outputs** (just under the scriptsig limit of 20kB!).

Using the Blocktrail Python SDK, I get a list of the outputs as hex using the following Python (2.7) code (NB, the *APIKEY*, *APISECRET* parameters are available if required from www.blocktrail.com):

```
from blocktrail import APIClient
bt_client = APIClient(APIKEY, APISECRET, network='BTC')
txnObj = bt_client.transaction('54e48e5f5c656b26c3bca14a8c95aa583d07ebe84dde3b7dd4a78f4e418
hashes = [(t['script_hex']) for t in (txnObj)['outputs']]
```

The resulting list is available here in full and is essentially all pay-to-pubkey-script Txns. An excerpt:

```
[u'5141e4cf0200067daf13255044462d312e340a25c3a4c3bcc3b6c39f0a322030206f626a0a3c3c2f4c656e677
.....
u'514130206e200a3030303031383235343020303030303030206e200a747261696c65720a3c3c2f53697a65203638
 u'51213e0a737461727478726566a3138323732370a2525454f460a0000000000000000051ae',
 u'76a91462e907b15cbf27d5425399ebf6f0fb50ebb88f1888ac',
 u'76a914031c79236ff3017496cf8d9a883f494458f245f288ac']
```

QUESTION: How is this array of hex data parsed into the bitcoin.pdf? Specific Python framed answers would be appreciated!

asked 2 years, 7 months ago

viewed 2,895 times

active 2 years, 6 months ago

P2SH - Really Flexible Addresses

- These are addresses that start with a “3”
- Flexible because part of the SCRIPT is kept by the creator of the address (RedeemScript)
- RedeemScript must be provided when spending
- RedeemScript is at first treated as an element, but then is interpreted as SCRIPT

P2SH - Really Flexible Addresses

Pay-to-Script-Hash

scriptPubKey (receiving)

`a91474d691da1574e6b3c192ecfb52cc8984ee7b6c5687`

```
a9 - OP_HASH160
14 - Length of <hash>
74d6...56 - <hash>
87 - OP_EQUAL
```

P2SH

scriptSig (spending)

```
00483045022100dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fe
f4f5dc0559bddfb94e02205a36d4e4e6c7fcd16658c50783e00c34160997
7aed3ad00937bf4ee942a8993701483045022100da6bee3c93766232079a
01639d07fa869598749729ae323eab8eef53577d611b02207bef15429dca
dce2121ea07f233115c6f09034c0be68db99980b9a6c5e75402201475221
022626e955ea6ea6d98850c994f9107b036b1334f18ca8830bfff1295d21
cfdb702103b287eaf122eea69030a0e9feed096bed8045c8b98bec453e1f
fac7fbdbd4bb7152ae
```

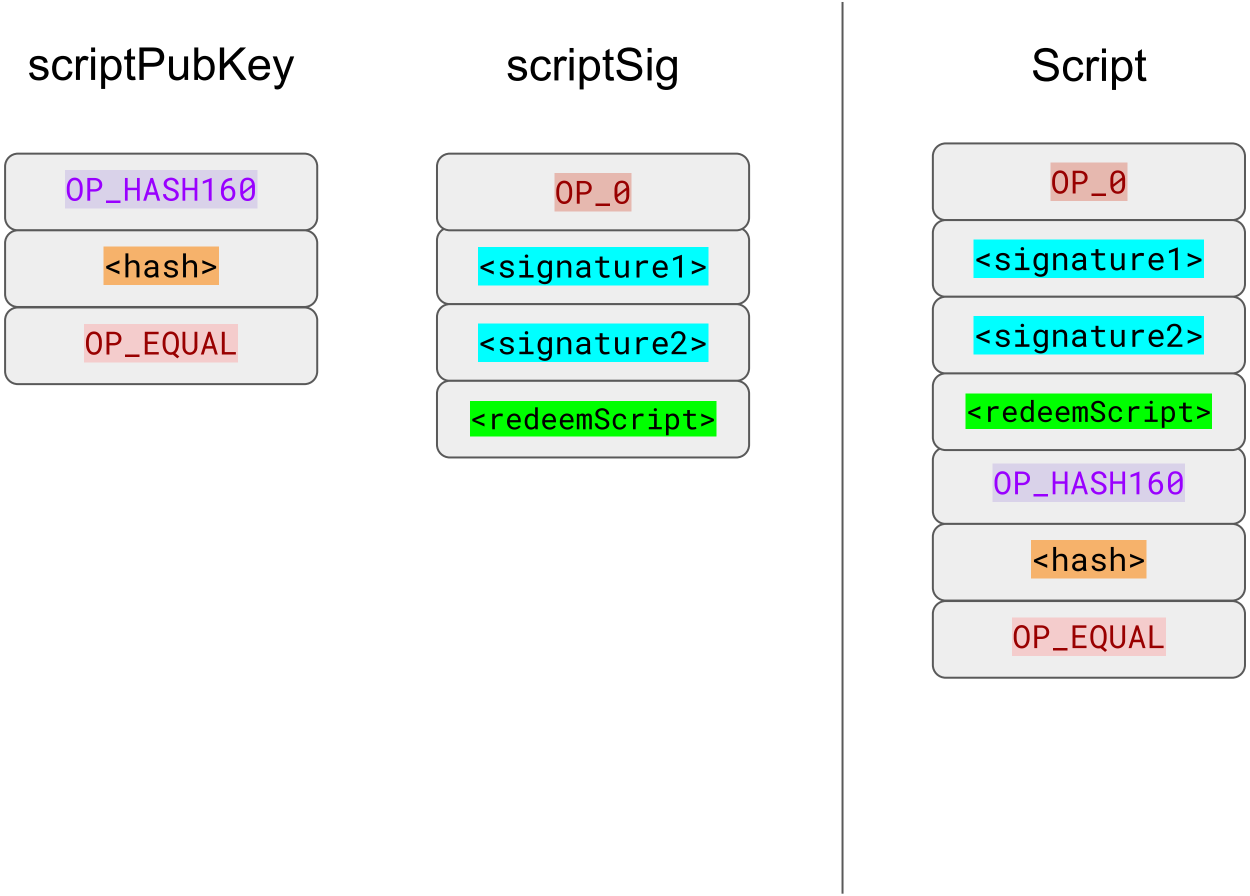
00 - OP_0

48 - Length of <signaturex>

3045...01 - <signaturex>

47 - Length of redeemScript

P2SH


scriptPubKey
OP_HASH160
<hash>
OP_EQUAL
scriptSig
OP_0
<signature1>
<signature2>
<redeemScript>
Script
OP_0
<signature1>
<signature2>
<redeemScript>
OP_HASH160
<hash>
OP_EQUAL

P2SH

Script

Stack

Processing

OP_0

<signature1>

<signature2>

<redeemScript>

OP_HASH160

<hash>

OP_EQUAL

P2SH

Script | Stack | Processing

`<signature1>`

`<signature2>`

`<redeemScript>`

`OP_HASH160`

`<hash>`

`OP_EQUAL`

Processing: `OP_0`

`OP_0` - Puts a 0 on the stack

P2SH

Script	Stack	Processing
`<signature1>`		`OP_0` - Puts a 0 on the stack
`<signature2>`		
`<redeemScript>`		
`OP_HASH160`		
`<hash>`		
`OP_EQUAL`	`0`	

P2SH

Script

OP_HASH160

<hash>

OP_EQUAL

Stack

<redeemScript>

<signature2>

<signature1>

0

Processing

P2SH

Script	Stack	Processing
	`<redeemScript>`	
	`<signature2>`	
`<hash>`	`<signature1>`	
`OP_EQUAL`	`0`	`OP_HASH160`

`OP_HASH160` - Does a SHA256 and then a RIPEMD160 to the top element.

P2SH

Script

<hash>

OP_EQUAL

Stack

<hash>

<signature2>

<signature1>

0

Processing

OP_HASH160 - Does a SHA256 and then a RIPEMD160 to the top element.

P2SH

Script

Stack

<hash>

<hash>

<signature2>

<signature1>

0

Processing

OP_EQUAL

OP_EQUAL - Checks that the top two elements are equal. If so, put a 1 on stack, if not put a 0 on stack

P2SH

Script

Stack

Processing

1

<signature2>

<signature1>

0

`OP_EQUAL` - Checks that the top two elements are equal. If so, put a 1 on stack, if not put a 0 on stack

P2SH

Script

Stack

1

<signature2>

<signature1>

0

Processing

Special rule: `OP_HASH160` `<hash>` `OP_EQUAL` evaluates to true means that the `<redeemScript>` is now evaluated as Script elements

P2SH

redeemScript

```
5221022626e955ea6ea6d98850c994f9107b036b1334f18ca8830bfff129
5d21cfdb702103b287eaf122eea69030a0e9feed096bed8045c8b98bec45
3e1ffac7fbdbd4bb7152ae
```

```
52 - OP_2
21 - Length of <pubkeyx>
0....01 - <pubkeyx>
52 - OP_2
ae - OP_CHECKMULTISIG
```

P2SH

Script

Stack

Processing

OP_2

<pubkey1>

<pubkey2>

OP_2

OP_CHECKMULTISIG

<signature2>

<signature1>

0

Special rule: OP_HASH160<hash> OP_EQUAL evaluates to true means that the <redeemScript> is now evaluated as Script elements

P2SH

Script	Stack	Processing
<pubkey1>	2	
<pubkey2>	<signature2>	
OP_2	<signature1>	
OP_CHECKMULTISIG	0	

P2SH

Script	Stack	Processing
	2	
	<pubkey2>	
	<pubkey1>	
	2	
	<signature2>	
	<signature1>	
OP_CHECKMULTISIG	0	

P2SH

Script

Stack

Processing

Stack
2
<pubkey2>
<pubkey1>
2
<signature2>
<signature1>
0

OP_CHECKMULTISIG

`OP_CHECKMULTISIG` - Checks if m of the signatures are valid of the n public keys for current transaction. Puts 1 back if valid, 0 otherwise.

P2SH

Script	Stack	Processing
	1	

`OP_CHECKMULTISIG` - Checks if m of the signatures are valid of the n public keys for current transaction. Puts 1 back if valid, 0 otherwise.

P2SH - Really Flexible Addresses

- RedeemScript was made new as a part of the introduction of p2sh (BIP0016)
- RedeemScript will be added to the processing queue only if the hash matches the hash in ScriptPubKey
- RedeemScript substitution is a bit hacky and was a huge controversy at the time.

P2SH Transaction

```
0100000001868278ed6ddfb6c1ed3ad5f8181eb0c7a385aa0836f01d5e4789e6bd304d87221a00
0000db0048304502210 0dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fef4f5dc0559bd
dfb94e02205a36d4e4e6c7fcd16658c50783e00c341609977aed3ad00937bf4ee942a899370148
3045022100da6bee3c93766232079a01639d07fa869598749729ae323eab8eef53577d611b0220
7bef15429dcadce2121ea07f233115c6f09034c0be68db99980b9a6c5e754022014752210 22626
e955ea6ea6d98850c994f9107b036b1334f18ca8830bfff1295d21cfdb702103b287eaf122eea6
9030a0e9feed096bed8045c8b98bec453e1ffac7fbdbd4bb7152aeffffffff04d3b11400000000
001976a914904a49878c0adfc3aa05de7afad2cc15f483a56a88ac7f400900000000001976a914
418327e3f3dda4cf5b9089325a4b95abdfa0334088ac722c0c00000000001976a914ba35042cfe
9fc66fd35ac2224eebdafd1028ad2788acdc4ace020000000017a91474d691da1574e6b3c192ec
fb52cc8984ee7b6c568700000000
```

01000000 - version

01 - # of inputs

8682...22 - previous tx hash

1a000000 - previous tx index

db00...ae - scriptSig

ffffffff - sequence

04 - # of outputs

d3b1...00 - output amounts

1976...ac - p2pkh scriptPubKey

17a9...87 - p2sh scriptPubKey

00000000 - locktime

Creating a p2sh address

`a91474d691da1574e6b3c192ecfb52cc8984ee7b6c5687`

`<hash> = 74d691da1574e6b3c192ecfb52cc8984ee7b6c56`

BIP0013 defines how to turn this into an address

For mainnet prepend byte `b'\x05'`, for testnet byte `b'\xc0'`

Example

```
from binascii import unhexlify
from helper import h160_to_p2sh_address

print(h160_to_p2sh_address(unhexlify('74d691da1574e6b3c192ec
fb52cc8984ee7b6c56'), testnet=False))
print(h160_to_p2sh_address(unhexlify('74d691da1574e6b3c192ec
fb52cc8984ee7b6c56'), testnet=True))
```

Study

```
helper.py:h160_to_p2sh_address
helper.py:HelperTest
```

Questions?

P2SH

scriptSig (spending)

```
00483045022100dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fef4f5dc0559bddfb94e
02205a36d4e4e6c7fcd16658c50783e00c341609977aed3ad00937bf4ee942a899370148304502
2100da6bee3c93766232079a01639d07fa869598749729ae323eab8eef53577d611b02207bef15
429dcadce2121ea07f233115c6f09034c0be68db99980b9a6c5e75402201475221022626e955ea
6ea6d98850c994f9107b036b1334f18ca8830bfff1295d21cfdb702103b287eaf122eea69030a0
e9feed096bed8045c8b98bec453e1ffac7fbdbd4bb7152ae
```

```
00 - OP_0
48 - Length of <signaturex>
3045...01 - <signaturex>
47 - Length of redeemScript
5221...ae - <redeemScript>
```

Verifying p2sh Transaction

```
Sig1 =
3045022100dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fef4f5
dc0559bddfb94e02205a36d4e4e6c7fcd16658c50783e00c341609977aed
3ad00937bf4ee942a8993701
Sig2 =
3045022100da6bee3c93766232079a01639d07fa869598749729ae323eab
8eef53577d611b02207bef15429dcadce2121ea07f233115c6f09034c0be
68db99980b9a6c5e75402201
```

What did this sign and how do we verify?

Verifying p2sh Transaction

```
0100000001868278ed6ddfb6c1ed3ad5f8181eb0c7a385aa0836f01d5e47
89e6bd304d87221a00000000ffffffff04d3b11400000000001976a91490
4a49878c0adfc3aa05de7afad2cc15f483a56a88ac7f400900000000001976
a914418327e3f3dda4cf5b9089325a4b95abdfa0334088ac722c0c0000
0000001976a914ba35042cfe9fc66fd35ac2224eebdafd1028ad2788acdc
4ace020000000017a91474d691da1574e6b3c192ecfb52cc8984ee7b6c56
8700000000
```

We replace the scriptSig for all inputs with 00, as before

Verifying p2sh Transaction

```
0100000001868278ed6ddfb6c1ed3ad5f8181eb0c7a385aa0836f01d5e47
89e6bd304d87221a000000475221022626e955ea6ea6d98850c994f9107b
036b1334f18ca8830bfff1295d21cfdb702103b287eaf122eea69030a0e9
feed096bed8045c8b98bec453e1ffac7fbdbd4bb7152aeffffffff04d3b1
140000000000001976a914904a49878c0adfc3aa05de7afad2cc15f483a56a
88ac7f400900000000001976a914418327e3f3dda4cf5b9089325a4b95ab
dfa0334088ac722c0c00000000001976a914ba35042cfe9fc66fd35ac222
4eebdafd1028ad2788acdc4ace020000000017a91474d691da1574e6b3c1
92ecfb52cc8984ee7b6c568700000000
```

`4752...ae - RedeemScript`

We replace the p2sh spending input with the RedeemScript instead of the scriptPubKey.

Verifying p2sh Transaction

```
0100000001868278ed6ddfb6c1ed3ad5f8181eb0c7a385aa0836f01d5e47
89e6bd304d87221a000000475221022626e955ea6ea6d98850c994f9107b
036b1334f18ca8830bfff1295d21cfdb702103b287eaf122eea69030a0e9
feed096bed8045c8b98bec453e1ffac7fbdbd4bb7152aeffffffff04d3b1
1400000000001976a914904a49878c0adfc3aa05de7afad2cc15f483a56a
88ac7f400900000000001976a914418327e3f3dda4cf5b9089325a4b95ab
dfa0334088ac722c0c00000000001976a914ba35042cfe9fc66fd35ac222
4eebdafd1028ad2788acdc4ace020000000017a91474d691da1574e6b3c1
92ecfb52cc8984ee7b6c56870000000001000000
```

01000000 - Sighash (SIGHASH_ALL)

Now we append the sigHash

Verifying p2sh Transaction

```
0100000001868278ed6ddfb6c1ed3ad5f8181eb0c7a385aa0836f01d5e4789e6bd304d87221a00
000047522102262 6e955ea6ea6d98850c994f9107b036b1334f18ca8830bfff1295d21cfdb7021
03b287eaf122eea69030a0e9feed096bed8045c8b98bec453e1ffac7fbdbd4bb7152aeffffffff
04d3b11400000000001976a914904a49878c0adfc3aa05de7afad2cc15f483a56a88ac7f400900
000000001976a914418327e3f3dda4cf5b9089325a4b95abdfa0334088ac722c0c000000000019
76a914ba35042cfe9fc66fd35ac2224eebdafd1028ad2788acdc4ace020000000017a91474d691
da1574e6b3c192ecfb52cc8984ee7b6c56870000000001000000
```

We now take the double_sha256 of the whole thing

```
from binascii import unhexlify
from helper import double_sha256

sha = double_sha256(unhexlify('0100...1000000'))
z = int.from_bytes(sha, 'big')
print(hex(z))

0xe71bfa115715d6fd33796948126f40a8cdd39f187e4afb03896795189fe1423c
```

P2SH

scriptSig (spending)

00483045022100dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fef4f5dc05
59bddfb94e02205a36d4e4e6c7fcd16658c50783e00c341609977aed3ad00937bf4e
e942a8993701483045022100da6bee3c93766232079a01639d07fa869598749729ae
323eab8eef53577d611b02207bef15429dcadce2121ea07f233115c6f09034c0be68
db99980b9a6c5e75402201475221022626e955ea6ea6d98850c994f9107b036b1334
f18ca8830bfff1295d21cfdb702103b287eaf122eea69030a0e9feed096bed8045c8
b98bec453e1ffac7fbdbd4bb7152ae

00 - OP_0
48 - Length of <signaturex>
3045...01 - <signaturex>
47 - Length of redeemScript
5221...ae - <redeemScript>

Verifying p2sh Transaction

We derive the r and s from the first signature

```
3045022100dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fef4f5
dc0559bddfb94e02205a36d4e4e6c7fcd16658c50783e00c341609977aed
3ad00937bf4ee942a8993701
```

- 30 - Marker
- 45 - Length of sig
- 02 - Marker for r value
- 21 - r value length
- 00dc…4e - r value
- 02 - Marker for s value
- 20 - s value length
- 5a36...37 - s value
- 01 - sighash

r = 0x00dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fef4f5dc0559bddfb94e

s = 0x5a36d4e4e6c7fcd16658c50783e00c341609977aed3ad00937bf4ee942a89937

P2SH

redeemScript

```
5221022626e955ea6ea6d98850c994f9107b036b1334f18ca8830bfff129
5d21cfdb702103b287eaf122eea69030a0e9feed096bed8045c8b98bec45
3e1ffac7fbdbd4bb7152ae
```

P2SH

redeemScript

`5221022626e955ea6ea6d98850c994f9107b036b1334f18ca8830bfff129`
`5d21cfdb702103b287eaf122eea69030a0e9feed096bed8045c8b98bec45`
`3e1ffac7fbdbd4bb7152ae`

`52 - OP_2`
`21 - Length of <pubkeyx>`
`0....01 - <pubkeyx>`
`52 - OP_2`
`ae - OP_CHECKMULTISIG`

Verifying p2sh Transaction

`022626e955ea6ea6d98850c994f9107b036b1334f18ca8830bfff1295d21cfdb70`

We derive the pubkey from the first pubkey

Example

```
from binascii import unhexlify
from io import BytesIO
from tx import Tx

raw_tx = BytesIO(unhexlify('01000000...0000000'))
tx_obj = Tx.parse(raw_tx)
for i, tx_in in enumerate(tx_obj.tx_ins):
    print(tx_obj.verify_input(i))
```

Study

`tx.py:Tx:verify_input`

`tx.py:TxTest:test_verify_input2`

Questions?