Transacciones de Bitcoin

Implementación

¿Cómo se ve una transacción?

Cuando la recibo por la red

Son bytes/hex

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

https://github.com/jimmysong/programmingbitcoin/blob/master/ch05.asciidoc

¿Cómo se ve una transacción?

Cuando la recibo por la red

Para procesar una transacción necesito:

- Serializar
- Deserializar (parse)

¿Qué contiene una transacción?

```
01000000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

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

Transacciones

Implementación

```
class Tx:

    '''
    What defines a transaction:
    1. Version
    2. Locktime
    3. Inputs
    4. Outputs
    5. Network (testnet or not)
    '''
    def __init__(self, version, tx_ins, tx_outs, locktime, testnet=False):
        self.version = version
        self.tx_ins = tx_ins
        self.tx_outs = tx_outs
        self.locktime = locktime
        self.testnet = testnet
```

Parsing

Implementación de transacciones

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

¿Cómo pasar de bytes a TxIn, TxOut, y Tx?

- Hay que implementar el método parse

Version

Parsing

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

- Version (4 bytes)

```python
# When we receive raw transaction (bytes) we want to parse it to figure out what's what
@classmethod
def parse(cls, serialization):
    # get the version:
    version = serialization[0:4]
```

(en realidad vamos a leer desde un stream; el número es little-endian; 1 = 0100000000)

Version

Parsing

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

```python
# When we receive raw transaction (bytes) we want to parse it to figure out what's what
@classmethod
def parse(cls, s, testnet=False):
    '''Takes a byte stream and parses the transaction at the start
    return a Tx object
    '''
    # s.read(n) will return n bytes
    # version is an integer in 4 bytes, little-endian
    version = little_endian_to_int(s.read(4))
```

Inputs

Parsing

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

El formato:

- Número de inputs = K
- input1, input2, …, inputK

Inputs

Parsing

Inputs/Outputs:

- Primer byte es número de inputs/outputs

¿Cuántos puedo tener?

1 byte = 8 bits = 2^8 = 256 = [0,255] mi rango

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Inputs

Parsing

Primer byte = número de inputs

- Max 255

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

¿Qué hago si quiero más?

- **Varint**

Varints

Parsing

Varint = variable integer

- Permite valores de $[0,2^{64}-1]$

Reglas varint(n):

1. $n < 253$: 1 byte = n
2. $253 <= n < 2^{16}-1$: 3 bytes: 253 (fd) + n (en 2 bytes little-endian)
3. $2^{16} <= n < 2^{32}-1$: 4 bytes: 254 (fe) + n (en 4 bytes little-endian)
4. $2^{32} <= n < 2^{64}-1$: 4 bytes: 255 (ff) + n (en 8 bytes little-endian)

100 = 0x64
255 = 0xfdff00
70015 = 0xfe7f110100

Varints

Parsing

```
def read_varint(s):
    '''read_varint reads a variable integer from a stream'''
    i = s.read(1)[0]
    if i == 0xfd:
        # 0xfd means the next two bytes are the number
        return little_endian_to_int(s.read(2))
    elif i == 0xfe:
        # 0xfe means the next four bytes are the number
        return little_endian_to_int(s.read(4))
    elif i == 0xff:
        # 0xff means the next eight bytes are the number
        return little_endian_to_int(s.read(8))
    else:
        # anything else is just the integer
        return i
```

Varints

Parsing

```
def encode_varint(i):
    '''encodes an integer as a varint'''
    if i < 0xfd:
        return bytes([i])
    elif i < 0x10000:
        return b'\xfd' + int_to_little_endian(i, 2)
    elif i < 0x100000000:
        return b'\xfe' + int_to_little_endian(i, 4)
    elif i < 0x10000000000000000:
        return b'\xff' + int_to_little_endian(i, 8)
    else:
        raise ValueError('integer too large: {}'.format(i))
```

Inputs

Parsing

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Elementos de un input:

- Previous transaction hash
- Previous input
- Scriptsig
- Sequence

Previous input, sequence little-endian; hash tb.

Inputs

Coding inputs

```
class TxIn:

    def __init__(self, prev_tx, prev_index, script_sig=None, sequence=0xffffffff):
        self.prev_tx = prev_tx
        self.prev_index = prev_index
        if script_sig is None:
            self.script_sig = Script()
        else:
            self.script_sig = script_sig
        self.sequence = sequence
```

Inputs

Coding inputs

```
class TxIn:

    def __init__(self, prev_tx, prev_index, script_sig=None, sequence=0xffffffff):
        self.prev_tx = prev_tx
        self.prev_index = prev_index
        if script_sig is None:
            self.script_sig = Script()
        else:
            self.script_sig = script_sig
        self.sequence = sequence
```

Nos falta explicar Script()

Inputs

Coding inputs

```
class TxIn:

    def __init__(self, prev_tx, prev_index, script_sig=None, sequence=0xffffffff):
        self.prev_tx = prev_tx
        self.prev_index = prev_index
        if script_sig is None:
            self.script_sig = Script()
        else:
            self.script_sig = script_sig
        self.sequence = sequence
```

Nos falta explicar sequence
(por ahora lo ignoraremos)

Inputs

Parsing

```
class TxIn:

    .
    .
    .

    @classmethod
    def parse(cls, s):
        '''Takes a byte stream and parses the tx_input at the start
        return a TxIn object
        '''
        # prev_tx is 32 bytes, little endian
        prev_tx = s.read(32)[::-1]
        # prev_index is an integer in 4 bytes, little endian
        prev_index = little_endian_to_int(s.read(4))
        # use Script.parse to get the ScriptSig
        script_sig = Script.parse(s)
        # sequence is an integer in 4 bytes, little-endian
        sequence = little_endian_to_int(s.read(4))
        # return an instance of the class (see __init__ for args)
        return cls(prev_tx, prev_index, script_sig, sequence)
```

Outputs

Parsing

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

El formato:

- Número de outputs = K (varint)
- output1, output2, …, outputK

Outputs

Parsing

¿Qué contiene un output?

- valor (8 bytes) 21,000,000 BTC = 2,100,000,000,000,000 Satoshis > 4 bytes
- ScriptPubKey

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Outputs

Implementación

```
class TxOut:

    def __init__(self, amount, script_pubkey):
        self.amount = amount
        self.script_pubkey = script_pubkey

    def __repr__(self):
        return '{}:{}'.format(self.amount, self.script_pubkey)

    @classmethod
    def parse(cls, s):
        '''Takes a byte stream and parses the tx_output at the start
        return a TxOut object
        '''
        # amount is an integer in 8 bytes, little endian
        amount = little_endian_to_int(s.read(8))
        # use Script.parse to get the ScriptPubKey
        script_pubkey = Script.parse(s)
        # return an instance of the class (see __init__ for args)
        return cls(amount, script_pubkey)
```

Transacciones

Implementación

```
class Tx:

    '''
    What defines a transaction:
    1. Version
    2. Locktime
    3. Inputs
    4. Outputs
    5. Network (testnet or not)
    '''
    def __init__(self, version, tx_ins, tx_outs, locktime, testnet=False):
        self.version = version
        self.tx_ins = tx_ins
        self.tx_outs = tx_outs
        self.locktime = locktime
        self.testnet = testnet
```

Transacciones

Implementación

```
class Tx:

    '''
    What defines a transaction:
    1. Version
    2. Locktime
    3. Inputs
    4. Outputs
    5. Network (testnet or not)
    '''
    def __init__(self, version, tx_ins, tx_outs, locktime, testnet=False):
        self.version = version
        self.tx_ins = tx_ins
        self.tx_outs = tx_outs
        self.locktime = locktime
        self.testnet = testnet
```

¿Cuándo la transacción puede entrar al blockchain = block number? (se ignora si sequence = 0xffffffff en todos los inputs)

Parsing a transaction

```python
class Tx:

    # When we receive raw transaction (bytes) we want to parse it to figure out what's what
    @classmethod
    def parse(cls, s, testnet=False):
        # version is an integer in 4 bytes, little-endian
        version = little_endian_to_int(s.read(4))
        # num_inputs is a varint, use read_varint(s)
        num_inputs = read_varint(s)
        # parse num_inputs number of TxIns
        inputs = []
        for _ in range(num_inputs):
            inputs.append(TxIn.parse(s))
        # num_outputs is a varint, use read_varint(s)
        num_outputs = read_varint(s)
        # parse num_outputs number of TxOuts
        outputs = []
        for _ in range(num_outputs):
            outputs.append(TxOut.parse(s))
        # locktime is an integer in 4 bytes, little-endian
        locktime = little_endian_to_int(s.read(4))
        # return an instance of the class (see __init__ for args)
        return cls(version, inputs, outputs, locktime, testnet=testnet)
```

¿Qué sabemos hasta ahora?

Parsing

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Cómo pasar de bytes a TxIn, TxOut, y Tx

Serialización

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

¿Cómo producir estos bytes?

- Serialización

Serialización

TxOut

```
class TxOut:

    ...

    def serialize(self):
        '''Returns the byte serialization of the transaction output'''
        # serialize amount, 8 bytes, little endian
        result = int_to_little_endian(self.amount, 8)
        # serialize the script_pubkey
        result += self.script_pubkey.serialize()
        return result
```

Serialización

TxIn

```
class TxIn:
    ...
    def serialize(self):
        '''Returns the byte serialization of the transaction input'''
        # serialize prev_tx, little endian
        result = self.prev_tx[::-1]
        # serialize prev_index, 4 bytes, little endian
        result += int_to_little_endian(self.prev_index, 4)
        # serialize the script_sig
        result += self.script_sig.serialize()
        # serialize sequence, 4 bytes, little endian
        result += int_to_little_endian(self.sequence, 4)
        return result
```

Serialización

Tx

```python
class Tx:
    ...
    # When we have a transaction object, to send it to the network we need to serialize it
    def serialize(self):
        '''Returns the byte serialization of the transaction'''
        # serialize version (4 bytes, little endian)
        result = int_to_little_endian(self.version, 4)
        # encode_varint on the number of inputs
        result += encode_varint(len(self.tx_ins))
        # iterate inputs
        for tx_in in self.tx_ins:
            # serialize each input
            result += tx_in.serialize()
        # encode_varint on the number of outputs
        result += encode_varint(len(self.tx_outs))
        # iterate outputs
        for tx_out in self.tx_outs:
            # serialize each output
            result += tx_out.serialize()
        # serialize locktime (4 bytes, little endian)
        result += int_to_little_endian(self.locktime, 4)
        return result
```

Clase Tx

Un método importante

```
class Tx:
    ...
    # Transaction ID; this is our hash pointer in the UTXO set
    # I.e. when you ask for a transaction to a full node, it guards it under this key
    def id(self):
        '''Human-readable hexadecimal of the transaction hash'''
        return self.hash().hex()

    # Well, just the hash
    # Note that the hash is given in "little-endian"
    def hash(self):
        '''Binary hash of the legacy serialization'''
        return hash256(self.serialize())[::-1]
```

Transacition fee

¿Qué nos falta?

Para trabajar con transacciones deberíamos poder:

- Conocer el valor de cada input

```
class TxIn:

    def __init__(self, prev_tx, prev_index, script_sig=None, sequence=0xffffffff):
        self.prev_tx = prev_tx
        self.prev_index = prev_index
        if script_sig is None:
            self.script_sig = Script()
        else:
            self.script_sig = script_sig
        self.sequence = sequence
```

Transacition fee

¿Qué nos falta?

Para trabajar con transacciones deberíamos poder:

- Conocer el valor de cada input
- Input = output de una transacción en el pasado

¿Cómo consigo este output?

- **De un full node!!!**

Show me the money

Pedir algo del UTXO de un nodo full

```
# The generic format (for mainnet):
#https://blockchain.info/rawtx/b6f6991d03df0e2e04dafffcd6bc418aac66049e2cd74b80f14ac86db1e3f0da?format=hex

#mainnet transaction:
tx_hash = 'b6f6991d03df0e2e04dafffcd6bc418aac66049e2cd74b80f14ac86db1e3f0da'

url = 'https://blockchain.info/rawtx/{}?format=hex'.format(tx_hash)
response = requests.get(url)

raw = bytes.fromhex(response.text)
stream = BytesIO(raw)
tx = Tx.parse(stream)
```

Show me the money

```python
class TxFetcher:

    @classmethod
    def get_url(cls, testnet=False):
        if testnet:
            return 'http://testnet.programmingbitcoin.com'
        else:
            return 'http://mainnet.programmingbitcoin.com'

    @classmethod
    def fetch(cls, tx_id, testnet=False):
        url = '{}/tx/{}.hex'.format(cls.get_url(testnet), tx_id)
        response = requests.get(url)
        try:
            raw = bytes.fromhex(response.text.strip())
        except ValueError:
            raise ValueError('unexpected response: {}'.format(response.text))
        # make sure the tx we got matches to the hash we requested
        if raw[4] == 0:
            # zero imputs = coinbase
            raw = raw[:4] + raw[6:]
            tx = Tx.parse(BytesIO(raw), testnet=testnet)
            tx.locktime = little_endian_to_int(raw[-4:])
        else:
            tx = Tx.parse(BytesIO(raw), testnet=testnet)
        if tx.id() != tx_id:
            raise ValueError('not the same id: {} vs {}'.format(tx.id(), tx_id))
        return tx
```

Transacition fee

¿Qué nos falta?

¿Cómo computo el txFee?

- sum = 0
- Para cada input
- Recupero la transacción del nodo full
- Pesco el output correspondiente
- Agrego el valor de este output a sum

Para los outputs más sencillo (ya tengo el valor)

Transacition fee

Ahora ustedes

```
'''
# The transaction class
'''
class Tx:

    ...

    # Computes the transaction fee
    # Recall that this will need to connect to a full node to get the input transaction!!!
    def fee(self):
        '''Returns the fee of this transaction in satoshi'''
        ######################
        ####IMPLEMENT THIS####
        ######################
        return 0
```

Transacition fee

Ahora ustedes

```
class TxIn:

    ...

    def value(self, testnet=False):
        '''Get the outpoint value by looking up the tx hash
        Returns the amount in satoshi
        '''
        ##########################
        ####IMPLEMENT THIS####
        ##########################
        # You will have to use the TxFetcher class
        return 0
```

Firmas

¿Qué uno firma en una transacción de Bitcoin?

- La transacción entera
- Con ScriptSig vacío

Firmas

¿Qué uno firma en una transacción de Bitcoin?

- La transacción entera
- Con ScriptSig ~~vacío~~ reemplazado por el ScriptPubKey que gasta
- Y un código de autorización (SIGHASH)

En realidad firmo hash de esto!

Firmas

¿Cómo genero el hash de lo que firmo en una transacción?

Una firma para cada input!

1. Saco todos los ScriptSig
2. Reemplazo mi input con su ScriptPubKey correspondente
3. Concateno un código especificando que se autoriza

Firmas

¿Qué firmo?

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

version
nr_inputs
prev_hash
prev_index
ScriptSig
sequence
…

Firmas – para cada input

Para cada input

Paso1:

- Sacar todos los ScriptSig

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
000feffffff02a135ef0100000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac
99c39800000000001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

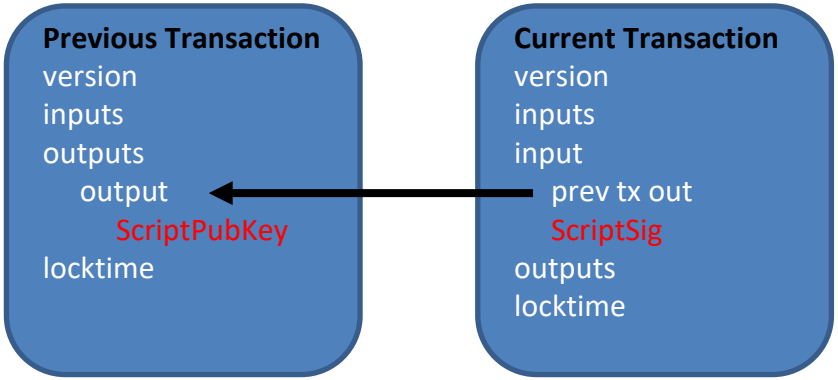
version
nr_inputs
prev_hash
prev_index
ScriptSig
sequence
…

Firmas – para cada input

Para cada input

Paso2:

- Reemplazar el ScriptSig del input que está firmando:

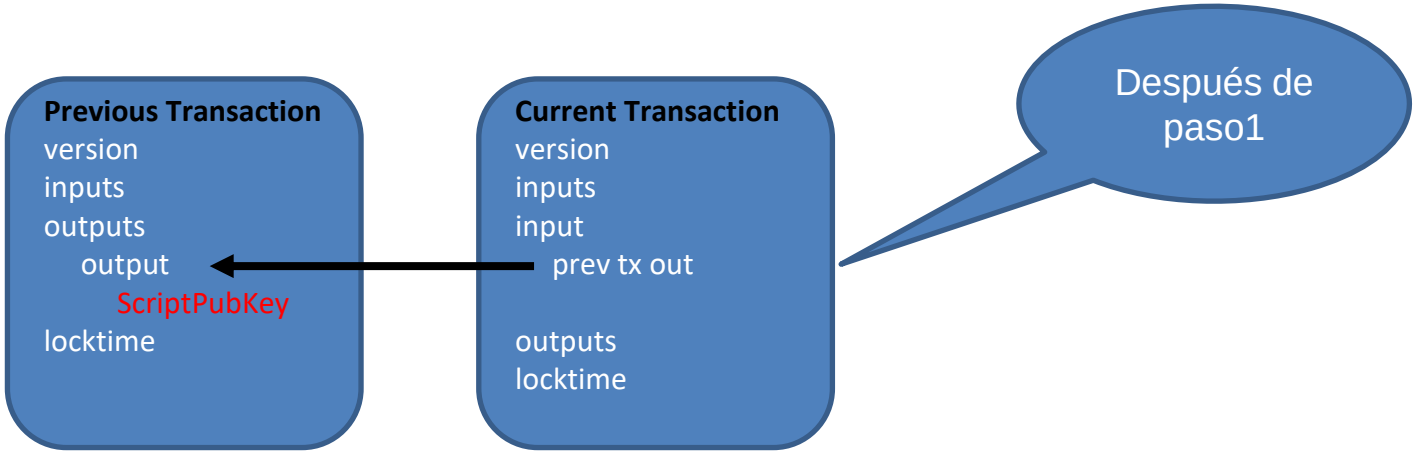


Firmas – para cada input

Para cada input

Paso2:

- Reemplazar el ScriptSig del input que está firmando:

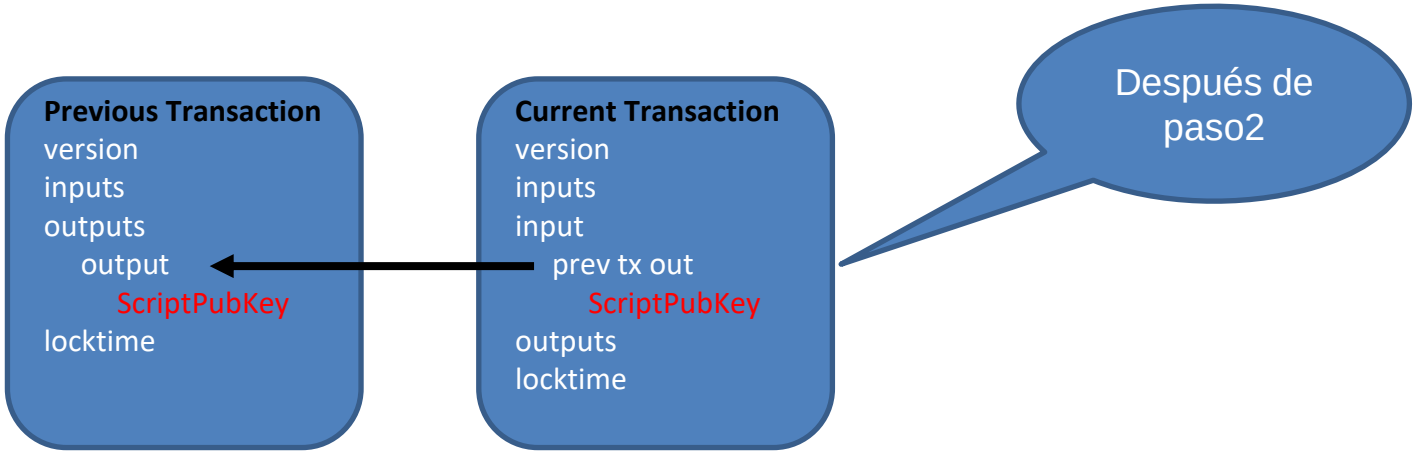


Firmas – para cada input

Para cada input

Paso2:

- Reemplazar el ScriptSig del input que está firmando:



Firmas – para cada input

Para cada input

Paso2:

- Reemplazar el ScriptSig del input que está firmando:

```
01000000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
000feffffff02a135ef0100000000 1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac
99c39800000000001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

version
nr_inputs
prev_hash
prev_index
ScriptSig
sequence
…

Firmas – para cada input

Para cada input

Paso2:

- Reemplazar el ScriptSig del input que está firmando:

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000)
01976a914a802fc56c704ce87c42d7c92eb75e7896bdc41ae88acfeffffff02a135ef010000000019:
76a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc76
2dd5423e332166702cb75f40df79fea1288ac19430600
```

version
nr_inputs
prev_hash
prev_index
ScriptSig
sequence
...

Firmas – para cada input

Para cada input

Paso2:

- Reemplazar el ScriptSig del input que está firmando

Firmas – para cada input

Para cada input

Paso2:

- Reemplazar el ScriptSig del input que está firmando
- **Si se trata de un P2SH!!!!!!!!!!!!!!!!!!!!!!!!!!!!!**
- ScriptSig se reemplaza por el <redeem script>
- Y **no** por el ScriptPubKey!!!

Firmas – para cada input

Para cada input

Paso3:

- Código de autorización (4 bytes) concatenado al final

SIGHASH_ALL -- input puede ir con otros inputs/outputs de la tx
SIGHASH_SINGLE – con un output especifico
SIGHASH_NONE – puede ir con cualquier output

SIGHASH_ALL = 1 en 4 bytes little-endian

Firmas – para cada input

Para cada input

Paso3:

- SIGHASH se concatena al final

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
01976a914a802fc56c704ce87c42d7c92eb75e7896bdc41ae88acfeffffff02a135ef0100000000 19
76a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc76
2dd5423e332166702cb75f40df79fea1288ac19430600 01000000
```

version
nr_inputs
prev_hash
prev_index
ScriptSig
sequence
...
SIGHASH_ALL

¿Qué firmo?

Para cada input

```python
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
    signed for index input_index'''
    # start the serialization with version
    # use int_to_little_endian in 4 bytes
    s = int_to_little_endian(self.version, 4)
    # add how many inputs there are using encode_varint
    s += encode_varint(len(self.tx_ins))
    # loop through each input using enumerate, so we have the input index
    for i, tx_in in enumerate(self.tx_ins):
        # if the input index is the one we're signing
        if i == input_index:
            # if the RedeemScript was passed in, that's the ScriptSig
            if redeem_script:
                script_sig = redeem_script
            # otherwise the previous tx's ScriptPubkey is the ScriptSig
            else:
                script_sig = tx_in.script_pubkey(self.testnet)
        # Otherwise, the ScriptSig is empty
        else:
            script_sig = None
        # add the serialization of the input with the ScriptSig we want
```

Version

¿Qué firmo?

Para cada input

```python
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
    signed for index input_index'''
    # start the serialization with version
    # use int_to_little_endian in 4 bytes
    s = int_to_little_endian(self.version, 4)
    # add how many inputs there are using encode_varint
    s += encode_varint(len(self.tx_ins))
    # loop through each input using enumerate, so we have the input index
    for i, tx_in in enumerate(self.tx_ins):
        # if the input index is the one we're signing
        if i == input_index:
            # if the RedeemScript was passed in, that's the ScriptSig
            if redeem_script:
                script_sig = redeem_script
            # otherwise the previous tx's ScriptPubkey is the ScriptSig
            else:
                script_sig = tx_in.script_pubkey(self.testnet)
        # Otherwise, the ScriptSig is empty
        else:
            script_sig = None
        # add the serialization of the input with the ScriptSig we want
```

Nr Inputs

¿Qué firmo?

Para cada input

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
    signed for index input_index'''
    # start the serialization with version
    # use int_to_little_endian in 4 bytes
    s = int_to_little_endian(self.version, 4)
    # add how many inputs there are using encode_varint
    s += encode_varint(len(self.tx_ins))
    # loop through each input using enumerate, so we have the input index
    for i, tx_in in enumerate(self.tx_ins):
        # if the input index is the one we're signing
        if i == input_index:
            # if the RedeemScript was passed in, that's the ScriptSig
            if redeem_script:
                script_sig = redeem_script
            # otherwise the previous tx's ScriptPubkey is the ScriptSig
            else:
                script_sig = tx_in.script_pubkey(self.testnet)
        # Otherwise, the ScriptSig is empty
        else:
            script_sig = None
        # add the serialization of the input with the ScriptSig we want
```

Iterar sobre inputs

¿Qué firmo?

Para cada input

```python
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
    signed for index input_index'''
    # start the serialization with version
    # use int_to_little_endian in 4 bytes
    s = int_to_little_endian(self.version, 4)
    # add how many inputs there are using encode_varint
    s += encode_varint(len(self.tx_ins))
    # loop through each input using enumerate, so we have the input ind
    for i, tx_in in enumerate(self.tx_ins):
        # if the input index is the one we're signing
        if i == input_index:
            # if the RedeemScript was passed in, that's the ScriptSig
            if redeem_script:
                script_sig = redeem_script
            # otherwise the previous tx's ScriptPubkey is the ScriptSig
            else:
                script_sig = tx_in.script_pubkey(self.testnet)
        # Otherwise, the ScriptSig is empty
        else:
            script_sig = None
        # add the serialization of the input with the ScriptSig we want
```

Si este es el input con cual firmo

¿Qué firmo?

Para cada input

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
    signed for index input_index'''
    # start the serialization with version
    # use int_to_little_endian in 4 bytes
    s = int_to_little_endian(self.version, 4)
    # add how many inputs there are using encode_varint
    s += encode_varint(len(self.tx_ins))
    # loop through each input using enumerate, so we have the input ind
    for i, tx_in in enumerate(self.tx_ins):
        # if the input index is the one we're signing
        if i == input_index:
            # if the RedeemScript was passed in                the ScriptSig
            if redeem_script:
                script_sig = redeem_script
            # otherwise the previous tx's ScriptPubkey is the ScriptSig
            else:
                script_sig = tx_in.script_pubkey(self.testnet)
        # Otherwise, the ScriptSig is empty
        else:
            script_sig = None
        # add the serialization of the input with the ScriptSig we want
```

P2SH

¿Qué firmo?

Para cada input

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
    signed for index input_index'''
    # start the serialization with version
    # use int_to_little_endian in 4 bytes
    s = int_to_little_endian(self.version, 4)
    # add how many inputs there are using encode_varint
    s += encode_varint(len(self.tx_ins))
    # loop through each input using enumerate, so we have the in
    for i, tx_in in enumerate(self.tx_ins):
        # if the input index is the one we're signing
        if i == input_index:
            # if the RedeemScript was passed in, that's the
            if redeem_script:
                script_sig = redeem_script
            # otherwise the previous tx's Scri          is the ScriptSig
            else:
                script_sig = tx_in.script_pubkey(self.testnet)
        # Otherwise, the ScriptSig is empty
        else:
            script_sig = None
        # add the serialization of the input with the ScriptSig we want
```

Else usa ScriptPubKey del output que se gasta

¿Qué firmo?

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_i
    '''Returns the integer
    signed for index input_
    # start the serializat
    # use int_to_little_en
    s = int_to_little_endi
    # add how many inputs
    s += encode_varint(len
    # loop through each in
    for i, tx_in in enumer
        # if the input ind
        if i == input_inde
            # if the Redee
            if redeem_scri
                script_sig
            # otherwise th
            else:
                script_sig = tx_in.script_pubkey(self.testnet)
        # Otherwise, the ScriptSig is empty
        else:
            script_sig = None
        # add the serialization of the input with the ScriptSig we want
```

```
class TxIn:

    ...

    def fetch_tx(self, testnet=False):
        return TxFetcher.fetch(self.prev_tx.hex(), testnet=testnet)

    def script_pubkey(self, testnet=False):
        '''Get the ScriptPubKey by looking up the tx hash
        Returns a Script object
        '''
        # use self.fetch_tx to get the transaction
        tx = self.fetch_tx(testnet=testnet)
        # get the output at self.prev_index
        # return the script_pubkey property
        return tx.tx_outs[self.prev_index].script_pubkey
```

¿Qué firmo?

Para cada input

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
    signed for index input_index'''
    # start the serialization with version
    # use int_to_little_endian in 4 bytes
    s = int_to_little_endian(self.version, 4)
    # add how many inputs there are using encode_varint
    s += encode_varint(len(self.tx_ins))
    # loop through each input using enumerate, so we have the in
    for i, tx_in in enumerate(self.tx_ins):
        # if the input index is the one we're signing
        if i == input_index:
            # if the RedeemScript was passed in, that's the
            if redeem_script:
                script_sig = redeem_script
            # otherwise the previous tx's ScriptPubke          criptSig
            else:
                script_sig = tx_in.script_pu          self.testnet)
        # Otherwise, the ScriptSig is em
        else:
            script_sig = None
        # add the serialization of the input with the ScriptSig we want
```

Si no es mi input script es vacio

¿Qué firmo?

Para cada input

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
        # add the serialization of the input with the ScriptSig we want
        s += TxIn(
            prev_tx=tx_in.prev_tx,
            prev_index=tx_in.prev_index,
            script_sig=script_sig,
            sequence=tx_in.sequence,
        ).serialize()
    # add how many outputs there are using encode_varint
    s += encode_varint(len(self.tx_outs))
    # add the serialization of each output
    for tx_out in self.tx_outs:
        s += tx_out.serialize()
    # add the locktime using int_to_little_endian in 4 bytes
    s += int_to_little_endian(self.locktime, 4)
    # add SIGHASH_ALL using int_to_little_endian in 4 bytes
    s += int_to_little_endian(SIGHASH_ALL, 4)
    # hash256 the serialization
    h256 = hash256(s)
    # convert the result to an integer using int.from_bytes(x, 'big')
    return int.from_bytes(h256, 'big')
```

Serializo el input

¿Qué firmo?

Para cada input

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
        # add the serialization of the input with the ScriptSig we want
        s += TxIn(
            prev_tx=tx_in.prev_tx,
            prev_index=tx_in.prev_index,
            script_sig=script_sig,
            sequence=tx_in.sequence,
        ).serialize()
    # add how many outputs there are using encode_varint
    s += encode_varint(len(self.tx_outs))
    # add the serialization of each output
    for tx_out in self.tx_outs:
        s += tx_out.serialize()
    # add the locktime using int_to_little_endian in 4 bytes
    s += int_to_little_endian(self.locktime, 4)
    # add SIGHASH_ALL using int_to_little_endian in 4 bytes
    s += int_to_little_endian(SIGHASH_ALL, 4)
    # hash256 the serialization
    h256 = hash256(s)
    # convert the result to an integer using int.from_bytes(x, 'big')
    return int.from_bytes(h256, 'big')
```

Nr outputs

¿Qué firmo?

Para cada input

```python
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
        # add the serialization of the input with the ScriptSig we want
        s += TxIn(
            prev_tx=tx_in.prev_tx,
            prev_index=tx_in.prev_index,
            script_sig=script_sig,
            sequence=tx_in.sequence,
        ).serialize()
    # add how many outputs there are using encode_varint
    s += encode_varint(len(self.tx_outs))
    # add the serialization of each output
    for tx_out in self.tx_outs:
        s += tx_out.serialize()
    # add the locktime using int_to_little_endian in 4 bytes
    s += int_to_little_endian(self.locktime, 4)
    # add SIGHASH_ALL using int_to_little_endian in 4 bytes
    s += int_to_little_endian(SIGHASH_ALL, 4)
    # hash256 the serialization
    h256 = hash256(s)
    # convert the result to an integer using int.from_bytes(x, 'big')
    return int.from_bytes(h256, 'big')
```

Serializo cada output

¿Qué firmo?

Para cada input

```python
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
        # add the serialization of the input with the ScriptSig we want
        s += TxIn(
            prev_tx=tx_in.prev_tx,
            prev_index=tx_in.prev_index,
            script_sig=script_sig,
            sequence=tx_in.sequence,
        ).serialize()
    # add how many outputs there are using encode_varint
    s += encode_varint(len(self.tx_outs))
    # add the serialization of each output
    for tx_out in self.tx_outs:
        s += tx_out.serialize()
    # add the locktime using int_to_little_endian in
    s += int_to_little_endian(self.locktime, 4)
    # add SIGHASH_ALL using int_to_little_endian in 4 bytes
    s += int_to_little_endian(SIGHASH_ALL, 4)
    # hash256 the serialization
    h256 = hash256(s)
    # convert the result to an integer using int.from_bytes(x, 'big')
    return int.from_bytes(h256, 'big')
```

Locktime

¿Qué firmo?

Para cada input

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
        # add the serialization of the input with the ScriptSig we want
        s += TxIn(
            prev_tx=tx_in.prev_tx,
            prev_index=tx_in.prev_index,
            script_sig=script_sig,
            sequence=tx_in.sequence,
        ).serialize()
    # add how many outputs there are using encode_varint
    s += encode_varint(len(self.tx_outs))
    # add the serialization of each output
    for tx_out in self.tx_outs:
        s += tx_out.serialize()
    # add the locktime using int_to_little_endian in 4 by
    s += int_to_little_endian(self.locktime, 4)
    # add SIGHASH_ALL using int_to_little_endian i      bytes
    s += int_to_little_endian(SIGHASH_ALL, 4)
    # hash256 the serialization
    h256 = hash256(s)
    # convert the result to an integer using int.from_bytes(x, 'big')
    return int.from_bytes(h256, 'big')
```

SIGHASH (4 bytes)

¿Qué firmo?

Para cada input

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
        # add the serialization of the input with the ScriptSig we want
        s += TxIn(
            prev_tx=tx_in.prev_tx,
            prev_index=tx_in.prev_index,
            script_sig=script_sig,
            sequence=tx_in.sequence,
        ).serialize()
    # add how many outputs there are using encode_varint
    s += encode_varint(len(self.tx_outs))
    # add the serialization of each output
    for tx_out in self.tx_outs:
        s += tx_out.serialize()
    # add the locktime using int_to_little_endian
    s += int_to_little_endian(self.locktime,
    # add SIGHASH_ALL using int_to_little_endian in 4 bytes
    s += int_to_little_endian(SIGHASH_ALL, 4)
    # hash256 the serialization
    h256 = hash256(s)
    # convert the result to an integer using int.from_bytes(x, 'big')
    return int.from_bytes(h256, 'big')
```

Doble SHA256

¿Qué firmo?

Para cada input

```
# Here we compute the serialization of what will be signed in a transaction
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
        # add the serialization of the input with the ScriptSig we want
        s += TxIn(
            prev_tx=tx_in.prev_tx,
            prev_index=tx_in.prev_index,
            script_sig=script_sig,
            sequence=tx_in.sequence,
        ).serialize()
    # add how many outputs there are using encode_varint
    s += encode_varint(len(self.tx_outs))
    # add the serialization of each output
    for tx_out in self.tx_outs:
        s += tx_out.serialize()
    # add the locktime using int_to_little_endian in 4 bytes
    s += int_to_little_endian(self.locktime, 4)
    # add SIGHASH_ALL using int_to_little_endian in 4 
    s += int_to_little_endian(SIGHASH_ALL, 4)
    # hash256 the serialization
    h256 = hash256(s)
    # convert the result to an integer using int.from_bytes(x, 'big')
    return int.from_bytes(h256, 'big')
```

Lo que firmo en el ECC

¿Cómo firmo?

Para cada input

Tengo mis bytes:

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
01976a914a802fc56c704ce87c42d7c92eb75e7896bdc41ae88acfeffffff02a135ef010000000019
76a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc76
2dd5423e332166702cb75f40df79fea1288ac19430600 01000000
```

```python
# Firmo para input 0
z = transaction.sig_hash(0)
private_key = PrivateKey(secret = 12345)
signature = private_key.sign(z).der()
signature = signature + SIGHASH_ALL.to_bytes(1,'big')
```

¿Cómo firmo?

Para cada input

Tengo mis bytes:

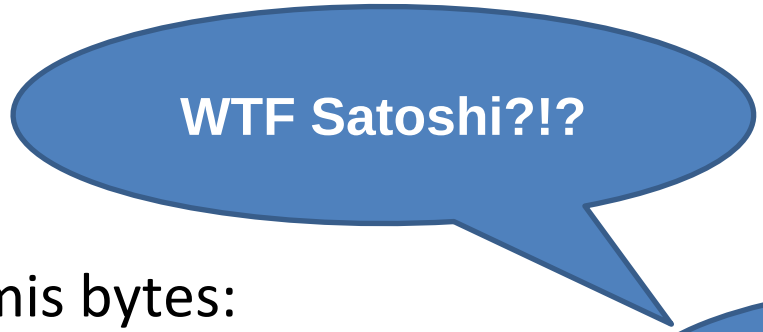
```
0100000001813f79011acb80925dfe69b3def35
01976a914a802fc56c704ce87c42d7c92eb75e78
76a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada
2dd5423e332166702cb75f40df79fea1288ac1943060001000000
```

1 byte!!!
Big-endian!!!

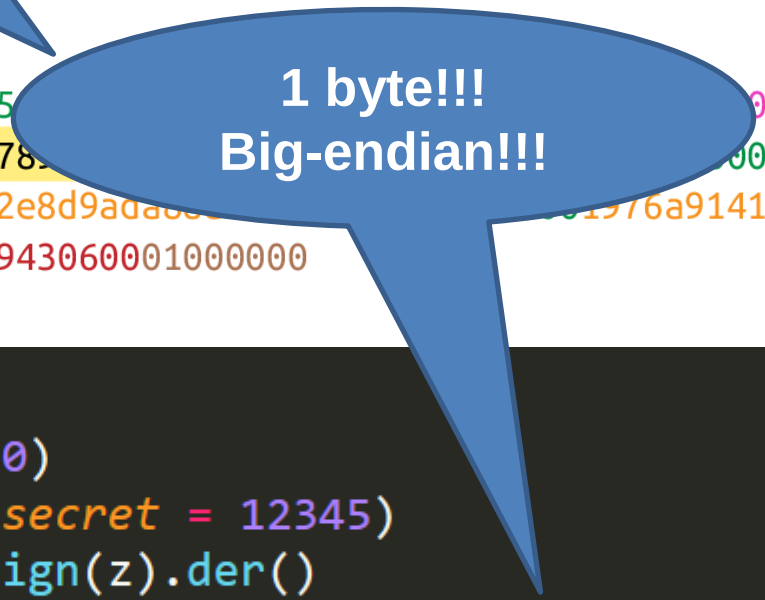
```python
# Firmo para input 0
z = transaction.sig_hash(0)
private_key = PrivateKey(secret = 12345)
signature = private_key.sign(z).der()
signature = signature + SIGHASH_ALL.to_bytes(1,'big')
```

¿Cómo firmo?

Para cada input



Tengo mis bytes:




```
# Firmo para input 0
z = transaction.sig_hash(0)
private_key = PrivateKey(secret = 12345)
signature = private_key.sign(z).der()
signature = signature + SIGHASH_ALL.to_bytes(1,'big')
```

Todo sobre firmas

Revisar:

- https://en.bitcoin.it/wiki/OP_CHECKSIG
- https://bitcoin.stackexchange.com/questions/3374/how-to-redeem-a-basic-tx

¿Qué nos falta?

Para poder crear transacciones necesitamos poder:

- Crear/serializar scripts de Script!!!

Para poder validar transacciones necesitamos poder:

- Correr los scripts de Script!!!

La clase que viene lo implementaremos

Referencias

- Jimmy Song, Programming Bitcoin, capítulos 5,6,7