Bitcoin and Blockchain Lab

In this lab, you will gain experience creating transactions using the Bitcoin testnet blockchain. The lab provides a starter code which uses python-bitcoinlib, a free python3 lib for manipulating bitcoin transaction. You need to fill the missing parts of the programs according to the problems stated below.

Testnet

Rather than having you download the entire testnet blockchain and run a bitcoin client on your machine, we will be making use of an online block explorer to upload and view transactions. The one that we will be using is called **BlockCypher**, which features a nice web interface as well as an API for submitting raw transactions that the starter code uses to broadcast the transactions you create for the exercises. After completing and running the code for each exercise, BlockCypher will return a JSON representation of your newly created transaction, which will be printed to your terminal. An example transaction object along with the meaning of each field can be found at BlockCypher's developer API documentation at

https://www.blockcypher.com/dev/bitcoin/#tx.

Of particular interest for the purposes of this lab will be the **hash**, **inputs**, and **outputs** fields. Note that you will use the **Bitcoin testnet** (the current version is Testnet3).

Script Opcodes

Your code will use the Bitcoin stack machine's opcodes, which are documented on the Bitcoin Wiki (**https://en.bitcoin.it/wiki/Script**). When composing programs for your transactions' scriptPubKeys and scriptSigs you may specify opcodes by using their names verbatim. Examples of some opcodes that you will likely be making use of include OP_DUP, OP_CHECKSIG, OP_EQUALVERIFY, and OP_CHECKMULTISIG, but you will end up using additional ones as well; check Wiki for more opcodes.

Setup. Unzip BTCLab.zip, you should see requirements.txt. Install bitcoin library, by running
$pip install –r requirements.txt

1 Generate key for you, Alice and Bob for using on **bitcoin testnet**:

Generate key pairs for you, Alice, and Bob. To do this, run **lib/keygen.py** to generate private keys for my_private_key, alice_secret_key_BTC
and bob_secret_key_BTC, and record these keys in **lib/config.py**.

2. Obtain some test bitcoins for **my_private_key** and **alice_secret_key_BTC**:

a. Go to the Bitcoin Testnet faucet (**https://testnet-faucet.com/btc-testnet/ or** https://coinfaucet.eu/en/btc-testnet/) and paste in the corresponding addresses of the users. Note that faucets will often rate-limit requests for coins based on Bitcoin

address and IP address, so try not to lose your test Bitcoin too often. Note that the faucet limits requests by the same IP address to one every 12 hours.

b. Record the transaction ID the faucet provides as you will need it later. Viewing the transaction in a block explorer (e.g. **https://live.blockcypher.com**/) will also let you know which output of the transaction corresponds to your address, and you will need this utxo index for the next step as well.

3. (**P2PKH Transaction**) Open Q1.py and complete the scripts labelled with TODOs to redeem an output you own and send it back to the faucet with a standard PayToPublicKeyHash transaction. The faucet address is already included in the starter code for you. Your functions should return a list consisting of only OP codes and parameters passed into the function.

4. For this question, we will generate a transaction that is dependent on some constants.

(a) Open Q2a.py. Generate a transaction from address of alice_public_key that can be redeemed by the solution (x; y) to the following system of two linear equations (N1, N2 are your selected numbers):

x + y = N1 and x-y = N2

Make sure you use OP_ADD and OP_SUB in your scriptPubKey. Check your scriptPubKey so that when given scriptSig, the evaluation in the end will leave **TRUE** on the stack.

(b) Open Q2b.py. Redeem the transaction you generated above. The redemption script should be as small as possible. That is, a valid scriptSig should consist of simply pushing two integers x and y to the stack.

5. (**P2MS Transaction)** Next, we will create a multi-sig transaction involving four parties.

(a) Open Q3a.py. Generate a multi-sig transaction involving four parties such that the transaction can be redeemed by the first party (bank) combined with any one of the 3 others (customers) but not by only the customers or only the bank. You may assume the role of the bank for this problem so that the bank's private key is my_private_key and the bank's public key is my_public_key. Generate the customers' keys using lib/keygen.py and paste them in Q3a.py.

(b) Open Q3b.py. Redeem the transaction and make sure that the scriptSig is as small as possible. You can use any legal combination of signatures to redeem the transaction but make sure that all combinations would have worked.

6. (**optional**) Implement how to evenly split a coin owned by an address into n values owned by the same address. Split this within one transaction. The starter code is lib/split_test_coins.py.