CS251: Cryptocurrencies and Blockchain Technologies **Fall 2019**

Programming Project #1

Due: 11:59pm on Mon., **Oct. 14, 2019**
Submit via Gradescope code: **MG7EP3**

In this project, you will gain experience creating transactions using Bitcoin and BlockCypher testnet blockchains. This project consists of 4 questions, each of which is explained below. The starter code we provide uses python-bitcoinlib, a free, low-level Python 3 library for manipulating Bitcoin transactions.

1 Project Background

1.1 Anatomy of a Bitcoin Transaction

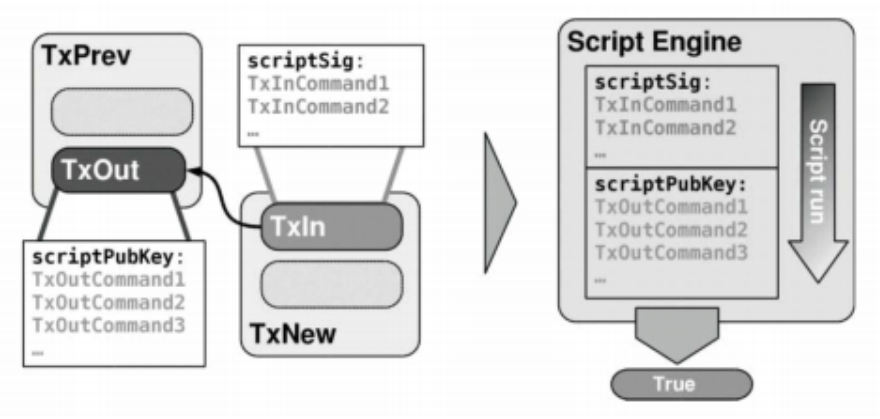

Figure 1: Each `TxIn` references the `TxOut` of a previous transaction, and a `TxIn` is only valid if its scriptSig outputs True when prepended to the `TxOut`'s scriptPubKey.

Bitcoin transactions are fundamentally a list of outputs, each of which is associated with an amount of bitcoin that is "locked" with a puzzle in the form of a program called a scriptPubKey (also sometimes called a "smart contract"), and a list of inputs, each of which references an output from the list of outputs and includes the "answer" to that output's puzzle in the form of a program called a scriptSig. Validating a scriptSig consists of appending the associated scriptPubKey to it, running the combined script and ensuring that it outputs True.

$$\text{run}(\textbf{ScripSig} \mid\mid \textbf{ScriptPK}) = \begin{cases} \text{True} & \text{valid ScripSig, TxIn spend TxOut} \\ \text{else} & \text{invalid ScripSig, TxIn cannot spend TxOut} \end{cases}$$

Most transactions are "PayToPublicKeyHash" or "P2PKH" transactions, where the scriptSig

is a list of the recipient's public key and signature, and the scriptPubKey performs cryptographic checks on those values to ensure that the public key hashes to the recipient's bitcoin address and the signature is valid.

Each transaction input is referred to as a `TxIn`, and each transaction output is referred to as a `TxOut`. The situation for a transaction with a single input and single output is summarized by Figure 1 above.

The sum of the bitcoin in the unspent outputs to a transaction must not exceed the sum of the inputs for the transaction to be valid. The difference between the total input and total output is implicitly taken to be a transaction fee, as a miner can modify a received transaction and add an output to their address to make up the difference before including it in a block.

$$\sum \text{TxIn} = \sum \text{TxOut} + \text{Tx}_{\text{fee}}$$

For the first 3 questions in this project, the transactions you create will consume one input and create one PayToPublicKeyHash output that sends an amount of bitcoin back to the testnet faucet. The 4th question will carry out a swap of coins between two entities, Alice and Bob. For these exercises, you will want to take the fee into account when specifying how much to send and subtract a bit from the amount in the output you're sending from, say .001 BTC. If you do not include a fee, it is likely that your transaction will never be added to the blockchain. Since BlockCypher (see Section 1.3) will delete transactions that remain unconfirmed after a day or two, it is very important that you include a fee to make sure that your transactions are eventually confirmed.

1.2 Script Opcodes

Your code will use the Bitcoin stack machine's opcodes, which are documented on the Bitcoin wiki [1]. When composing programs for your transactions' scriptPubKeys and scriptSigs you may specify opcodes by using their names verbatim. For example, below is an example of a function that returns a scriptPubKey that cannot be spent, but rather acts as storage space for an arbitrary piece of data that someone may want to save to the blockchain using the `OP_RETURN` opcode.

```
def save_message_scriptPubKey(message):
    return [OP_RETURN,
            message]
```

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.

1.3 Overview of Testnets

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 project will be the `hash`, `input`, and `output` fields.

Note that you will be using the Bitcoin test network (testnet) for questions 1-4 and the BlockCypher test network for question 4. These will be useful in testing your code. As part of these exercises, you will request coins to some addresses (more details below).

2 Getting Started

1. Download the starter code from the course website, navigate to the directory and run `pip install -r requirements.txt` to intall the required dependencies. Make sure that you are using Python 3.

2. Make sure you've understood the structure of Bitcoin transactions and read the references in the Recommended Reading section if you would like more information.

3. Read over the starter code. Here is a summary of what each of the files contain:

   `lib/keygen.py`:

   You will run this script to generate new private keys and corresponding addresses for the Bitcoin Testnet. Questions 1-3 will solely use these private keys, while question 4 will also require you to use an alternative method to generate Block Cypher Testnet keys. **You are not expected to modify this file.**

   `lib/split_test_coins.py`:

   You will run this script to split your coins across multiple unspent transaction outputs (UTXOs). You will have to edit this file to input details about which transaction output you are splitting, the UTXO index, etc.

   `lib/config.py`:

   You will modify this file to include the private keys for your users. Note that `my_private_key`, `alice_secret_key_BTC` and `bob_secret_key_BTC` will be generated using the `keygen.py` file. You will make web requests to generate `alice_secret_key_BCY` and `bob_secret_key_BCY`. Read the comments in `config.py` for how to do this.

   `lib/utils.py`:

   Contains various util methods. **You are not expected to modify this file.**

   `Q1.py, Q2a.y, Q2b.py, Q3a.py, Q3b.py, Q4.py`:

   You will have to modify the various scriptSig and scriptPubKey methods, as well as fill the transaction parameters. Note that for question 3, you will have to generate additional private and public keys for customers using the keygen.py file.

   `alice.py, bob.py`:

   Creates and submits transactions for Q4 on behalf of Alice and Bob. **You are not expected to modify these files.**

   `swap.py`

Contains the logic to carry out the atomic swap. **You are not expected to modify this file.**

`docs/transactions.py`

You are expected to fill this file with the transaction ids generated for questions 1-3.

`docs/Q4design_doc.txt`

You are expected to fill this design doc to explain your solution to Q4.

3 Setup

1. Open `lib/config.py` and read the file. Note that there are several users that you will need to generate private keys and addresses for.

2. First we are going to generate key pairs for you, Alice and Bob on the Bitcoin Testnet. Using `lib/keygen.py`, generate private keys for `my_private_key`, `alice_secret_key_BTC` and `bob_secret_key_BTC`, and record these keys in `lib/config.py`. Note that Alice and Bob's keys will only come into play for question 4. Please make sure to create different keys for Alice and Bob, you wouldn't want them to be able to forge each others' transactions!

3. Next, we want to get some test coins for `my_private_key` and `alice_secret_key_BTC`. To do so:

   (a) Go to the Bitcoin Testnet faucet (https://coinfaucet.eu/en/btc-testnet/), or to (https://bitcoinfaucet.uo1.net), 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.

   (b) Record the transaction hash the faucet provides, as you will need it for the next step. 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.

4. The faucet will give you and Alice one spendable output per person, but we would like to have multiple outputs to spend, at least 3 per exercise and preferably more in case we accidentally lock some with invalid scripts. Edit the parameters at the bottom of `split_test_coins.py`, where `txid` is the transaction hash of the faucet transaction from the previous step, `utxo_id` is 0 if your output was first in the faucet transaction and 1 if it was second, and `n` is the number of outputs you want your test coins split evenly into, and run the program with `python split_test_coins.py`. A perfect run through of questions 1-3 would require $n = 3$ for your address, one for each exercise, but if you anticipate accidentally locking an output due to a faulty script a couple times per exercise then you might want to set `n` to something higher like 8 so that you don't have to wait to access the faucet again or have to try with a different Bitcoin address. Similarly, you should split Alice's coins into multiple outputs just to be safe.

5. If `split_test_coins.py` was successful, you should get back some information about the transaction. Record the transaction hash, as each exercise will be spending an output from this transaction and will refer to it using this hash.

6. Next, we are going to create generate key pairs for Alice and Bob on the BlockCypher testnet.

   (a) Sign up for an account with Blockcypher to get an API token here: https://accounts.blockcypher.com/

   (b) Create BCY testnet keys for Alice and Bob and place into `lib/config.py`.

   ```
   curl -X POST https://api.blockcypher.com/v1/bcy/test/addrs?token=$YOURTOKEN
   ```

7. Give Bob's address bitcoin on the Blockcypher testnet (BCY).

   ```
   curl -d '{"address": "BOBS_BCY_ADDRESS", "amount": 1000000}' \
   https://api.blockcypher.com/v1/bcy/test/faucet?token=$YOURTOKEN
   ```

8. Let's also split Bob's coins using `split_test_coins.py`. Make sure to edit the parameters at the bottom of the file. Each time you are switching between the Bitcoin and BlockCypher testnets, make sure to visit `lib/config.py` and adjust the `network_type` variable. Make sure to record the transaction hash.

9. At the end, verify that you created Bitcoin Testnet addresses for you, Alice and Bob. You and Alice should have some coins on this blockchain. There should also be BlockCypher Testnet addresses for Alice and Bob. Bob should have some coins on this blockchain. Give yourself a pat on the back for finishing a long setup.

4 Questions

For each of the questions below, you will use the bitcoin script opcodes to create transactions. For question 4, you will write an atomic swap transaction across two different blockchains. To publish each transaction created for the exercises, edit the parameters at the bottom of the file to specify which transaction output the solution should be run with along with the amount to send in the transaction. If the scripts you write aren't valid, an exception will be thrown before they're published. For questions 1-3, make sure to record the transaction hash of the created transaction and write it to `transactions.py`. After completing each exercise, look up the transaction hash in a blockchain explorer to verify whether the transaction was picked up by the network. Make sure that all your transactions have been posted successfully before submitting their hashes.

Exercise 1. Open `Q1.py` and complete the scripts labelled with `TODO`s to redeem an output you own and send it back to the faucet with a standard PayToPublicKeyHash transaction. Your functions should return a list consisting of only OP codes and parameters passed into the function.

Exercise 2. For question 2, we will generate a transaction that is dependent on some constants.

(a) Open `Q2a.py`. Generate a transaction that can be redeemed by the solution (x, y) to the following system of two linear equations:

$$x + y = \text{(first half of your suid)} \qquad \text{and} \qquad x - y = \text{(second half or your suid)}$$

To ensure that an integer solution exists, please change the last digit of the two numbers on the right hand side so the numbers are both even or both odd.

(b) Open Q2b.py. Redeem the transaction. 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. Make sure you use OP_ADD and OP_SUB in your scriptPubKey.

Exercise 3. 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 your private key and the bank's public key is your public key. Generate the customers' keys using `keygen.py` and paste them in `Q3a.py`.

(b) Open `Q3b.py`. Redeem the transaction and make sure that the scriptPubKey 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.

Exercise 4. Last but not least, you will create a transaction called a *cross-chain atomic swap*, allowing two entities to securely trade ownership over cryptocurrencies on different blockchains. In this case, Alice and Bob will swap coins between the Bitcoin testnet and BlockCypher testnet. As you recall from setup, Alice has bitcoin on BTC Testnet3, and Bob has bitcoin on BCY Testnet. They want to trade ownership of their respective coins securely, something that can't be done with a simple transaction because they are on different blockchains. The idea here is to set up transactions around a secret x, that only one party (Alice) knows. In these transactions only H(x) will be published, leaving x secret. Transactions will be set up in such a way that once x is revealed, both parties can redeem the coins sent by the other party. If x is never revealed, both parties will be able to retrieve their original coins safely, without help from the other. Before you start, make sure to read `swap.py`, `alice.py`, and `bob.py`. Compare to the pseudocode in https://en.bitcoin.it/wiki/Atomic_cross-chain_trading. This will be very helpful in understanding this assignment. Note that for this question, you will only be editing `Q4.py` and you can test your code by running `python swap.py`.

(a) Consider the ScriptPubKey necessary for creating a transaction necessary for a cross-chain atomic swap. This transaction must be redeemable by the recipient (if they have a secret x that corresponds to Hash(x)), or redeemable with signatures from both the sender and the recipient. Write this ScriptPubKey in coinExchangeScript in `Q4.py`.

(b) Write the accompanying ScriptSigs:

i. Write the ScriptSig necessary to redeem the transaction in the case where the recipient knows the secret x. Write this in coinExchangeScriptSig1 in `Q4.py`.

ii. Write the ScriptSig necessary to redeem the transaction in the case where both the sender and the recipient sign the transaction. Write this in coinExchangeScriptSig2 in `Q4.py`.

(c) Run your code using `python swap.py`. We aren't requiring that the transactions be broadcasted, as that requires some waiting to validate transactions. Running with broadcast_transactions=False will validate that ScriptSig + ScriptPK return true. Try this for alice_redeems=True as well as alice_redeems=False.

OPTIONAL: Try with broadcast_transactions=True, which will make the code sleep for an appropriate amount of time to post everything to the blockchain and verify correctly. Warning: will take 30-60 minutes to run.

5 Submitting your code

Record your transaction hashes in the `transactions.py` file for questions 1-3. The hashes should be and listed one per line in the same order as the questions.

For question 4, make sure `Q4design_doc.txt` is filled out and your code verifies when run with `broadcast_transactions=False`. Your design doc should answer the following questions:

1. An explanation of what you wrote and how the coinExchangeScript work.
2. Briefly, how the coinExchangeScript you wrote fits into the bigger picture of this atomic swap.
3. Consider the case of Alice sending coins to Bob with coinExchangeScript:

   Why can Alice always get her money back if Bob doesn't redeem it?

   Why can't this be solved with a simple 1-of-2 multisig?

Please submit all code for this assignment. Please create a single .tar or .zip file that includes all your deliverables for all four questions. Submit via Gradescope using the code **MG7EP3**.

6 Recommended Reading

1. Bitcoin Script: https://en.bitcoin.it/wiki/Script
2. Bitcoin Transaction Format: https://en.bitcoin.it/wiki/Transaction
3. Bitcoin Transaction Details: https://privatekeys.org/2018/04/17/anatomy-of-a-bitcoin-transaction/
4. How Atomic Swap Works: https://en.bitcoin.it/wiki/Atomic_cross-chain_trading