Cryptography: Projects

(Deadline: 10am, Jan. 17, 2024)

Choose **two projects** from Project One, Project Two and Project Three. Submit your source codes (You may paste the codes into a pdf file). If you answered all three questions, TAs will arbitrarily choose two and grade them.

1 Project One (50 points)

In this project, you will implement the garbled circuit based protocol for computing $\mathbf{GE}(a, b)$, where $a = a_1a_0 \in \{0,1\}^2$ and $b = b_1b_0 \in \{0,1\}^2$. The implementation should contain

- a procedure that takes a boolean circuit of $\mathbf{GE}(a, b)$ as input and outputs a garbled circuit of $\mathbf{GE}(a, b)$; and
- a procedure that takes a garbled circuit of $\mathbf{GE}(a, b)$ and a set of input labels as input, evaluates the garbled circuit, and produces an output label.

For simplicity, you **do not need to implement the OT in the protocol**. Instead, you may simply ask Alice to send the labels $k_1^{b_1}, k_2^{b_1}$ and $k_5^{b_0}$ to Bob.

Hints. Given a length-preserving PRF $H : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$, the function $G : \{0,1\}^n \to \{0,1\}^{2n}$ defined by $G(k) = H_k(1)\|H_k(2)$ is a length-doubling PRG. You may define a length-doubling PRF $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$ in the following way:

$$F_k(x) = G(H_k(x)), \forall k, x \in \{0,1\}^n.$$

You may choose DES ($n = 64$) or AES ($n = 128$) as the PRF H. **You need to implement the PRF**.

2 Project Two (50 points)

In this project you will implement a Python function to generate a Merkle proof. Please download the starter code in `proj2.zip`. In it you will find four files:

- `prover.py`: This script generates the arguments for the verfier and writes them to a file for the verifier to read. Specifically it writes the leaf position, the leaf value and finally the hashes used to prove of the leafs presence at the given position in the Merkle tree.

  Start python script by running `python3 prover.py 683` from the command line. This script first calls the function `gen_leaves_for_merkle_tree()` to generate a thousand strings that will make up the leaves of a Merkle tree. Next it calls the method `gen_merkle_proof()` to generate the hashes for the Merkle proof for leaf number 683 (683 is the number provided on the command line). Finally, it writes the Merkle proof to a text file `merkle_proof.txt`.

  Your job is to implement the function `gen_merkle_proof()`. The missing code in that function can be implemented in less than ten lines of Python.

- `verifier.py`: This script contains a hardcoded Merkle root `ROOT` for the Merkle tree whose leaves were generated by `gen_leaves_for_merkle_tree()`. The script reads in the Merkle proof generated by the prover and verifies that the proof proves the leaf is at the stated position with respect to the hardcoded `ROOT`. **You should not make any changes to this file.** However, you should familiarize yourself with the function `compute_merkle_root_from_proof()`, which is the core of the verifier. This will help you implement the function `gen_merkle_proof()`. Your prover will need to generate a Merkle proof that is accepted by this verifier.

- `merkle_utils.py`: This python script contains helpers to make generating the proof and verifying the proof easier. **You should not make any changes to this file.** However, **you must** familiarise yourself with what each helper function and class does to be able to understand the starter code in `prover.py` and `verifier.py`.

- `proof-for-leaf-95.txt`: An example Merkle proof for leaf #95. Your task is to generate a proof file like this for the leaf specified on the command line.

After you implement the function `gen_merkle_proof()` in `prover.py`, running both scripts one after the other should generate the following output:

```
$ python3 prover.py 683

          I generated 1000 leaves for a Merkle tree of height 10.

          I generated a Merkle proof for leaf #683 in file merkle_proof.txt

$ python3 verifier.py 683

  I verified the Merkle proof:  leaf #683 in the committed tree is "data item 683".
```

Try changing one character in `merkle_proof.txt` and check that the verifier now rejects the proof. Note: there are many implementations of Merkle trees on the web. However, it is more fun, and much more instructive, to implement the prover yourself.

Hints. To aid your understanding of the starter code, try to answer the following questions for your self:

- What does the verifier expect you to include in the `proof`?
- How is `height` defined?
- What is the purpose of the `padding` in `gen_merkle_proof()`?

You do not need to submit your answers to these questions.

Another helpful exercise is to generate the Merkle proof for leaf number 95 with the command `python3 prover.py 95` so you can compare the result to the expected output provided in the file `proof-for-leaf-95.txt`.

Additional reading. We discussed Merkle trees in class, but if you want to read more about them, then this is a good resource.

3 Project Three (50 points)

In this project, you will gain experience creating transactions using the Bitcoin and BlockCypher testnet blockchains and Bitcoin Script. 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.

3.1 Project Background

3.1.1 Anatomy of a Bitcoin Transaction

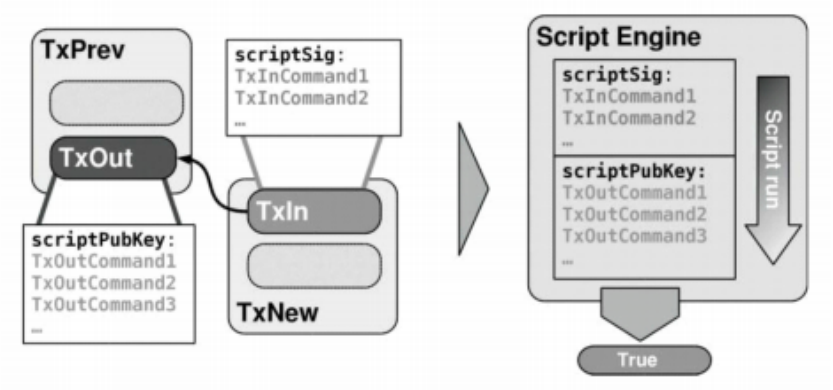

Figure 1: Each `TxIn` references the `TxOut` of a previous transaction, and a `TxIn` is only valid if its scriptSig outputs True when prepended to the `TxOut`'s scriptPubKey.

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.

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

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, say 0.001 BTC (this is just to be safe, you can probably include a fee as low as 0.00001 BTC if your funds are running low). If you do not include a fee, it is likely that your transaction will never be added to the blockchain. Since BlockCypher (see Section 3.1.3) 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.

3.1.2 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.

3.1.3 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`, `inputs`, and `outputs` fields. Note that you will be using two different test networks ("testnets") for this project: the Bitcoin testnet (the current version is Testnet3) for questions 1-4 and the BlockCypher testnet 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).

3.2 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. For this project, ensure that you are using Python 3. If you are not using a Python virtual environment, you must do two things differently. First, use `pip3` instead of `pip` to install packages to Python 3. Second, use the `python3` command to run scripts instead of `python` to run with the Python 3 interpreter.

2. Make sure you understand the structure of Bitcoin transactions and read the references in the Recommended Reading section below 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 `lib/keygen.py` file. You will make web requests to generate `alice_secret_key_BCY` and `bob_secret_key_BCY`. There are comments in `config.py` and instructions during setup 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 `lib/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.

4. Be sure to start early on this project, as block confirmation times can vary depending on how busy the network is!

3.3 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. 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`. 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) Use the following Testnet faucet to generate test BTC. (https://testnet-faucet.com/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. It is recommended that you use the address associated with `my_private_key` with the first faucet listed above since that faucet gives more coins and you will be performing more exercises with that address. Note that if the faucet is used more, it will take more time to receive the coins, so it is important to start the project early! Note that the faucet limits requests by the same IP address to one every hour.
   (b) Record the transaction hash 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. If the faucet doesn't give you a transaction hash, you can also paste the user address into the block explorer and find the transaction that way.

4. 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'
   ```

   Note, if you copy this command directly into your terminal from this handout, you'll likely need to delete and retype the `'` for the command to work.

5. Give Bob's address bitcoin on the Blockcypher testnet (BCY) and record the transaction hash.

   ```
   curl -d '{"address": "BOBS_BCY_ADDRESS", "amount": 100000}' \
   https://api.blockcypher.com/v1/bcy/test/faucet?token=YOURTOKEN
   ```

   Note, if you copy this command directly into your terminal from this handout, you'll likely need to delete and retype the `'{` and the `}'`, delete the `\`, and condense the command into one line for it to work.

6. The faucets will give you, Alice, and Bob one spendable output per person, but we would like to have multiple outputs to spend in case we accidentally lock some with invalid scripts. Edit the parameters at the bottom of `split_test_coins.py`, where `txid_to_spend` is the transaction hash from the faucet to your address, `utxo_index` 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. 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.

   Note: The faucet transaction would need to be fully verified (at least 6/6 confirmations) before you can split the coins you received. Waiting times will vary based on how busy the network is.

7. You should also split Alice's and Bob's coins into multiple outputs just to be safe. Note that each time you switch between the Bitcoin and BlockCypher tesntets, you should adjust the `network_type` variable in `lib/config.py`.

8. 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. Now it's time to explore creating transactions with Bitcoin Script.

3.4 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 `docs/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. 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.

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:

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

For an integer solution to exist, the rightmost digit of the first and second halves of your suid must either be both even or both odd. Therefore, you can change the rightmost digit of the second half of your suid to match the evenness or oddness of the righmost digit of the first half. Make sure you use `OP_ADD` and `OP_SUB` in your scriptPubKey.

(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.

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 `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.

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 the 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 to carry out 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 or more minutes to run.

(d) Fill in `docs/Q4design_doc.txt` with the following information:

i. An explanation of what you wrote and how the coinExchangeScript works.

ii. Briefly, how the coinExchangeScript you wrote fits into the bigger picture of this atomic swap.

iii. 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?

3.5 Submitting your code

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

For question 4, make sure `docs/Q4design_doc.txt` is filled out and your code verifies when run with `broadcast_transactions=False`.

3.6 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