Assignment 2

CS165: Computer Security,
Fall 2016, UC Riverside

Handed out: 11/18/2016
Due: 11/30/2016, 11:59 pm

Instructions:
1. You're allowed to work in groups of two.

2. Create a folder with your code, add a README file with the the details of the team and write down the individual contributions of each team member, add a RESULTS.txt file with results of both the parts, compress it (Zip/Tar.gz) name it CS165-A2-NetID1-NedID2, with the appropriate file extension, and submit it via iLearn.

3. You must abide by the UCR Academic Integrity Code. Your code will be checked and any violations will lead to immediate disqualification. Please find a copy of the Code at:
http://conduct.ucr.edu/docs/academicintegritybrochurestudent.pdf

4. You will require some familiarity with Java. There's a good set of tutorials at:
https://docs.oracle.com/javase/tutorial/

5. Acknowledgement - This assignment is based on the assignments given out in course COS 432 by Ed Felton et al. at Princeton.

6. ***You will be required to tell us about your contribution to the project and you will be graded on the basis of it.***

Start with this: https://www.youtube.com/channel/UCNcSSleedtfyDuhBvOQzFzQ

Part A - Verifying Transactions

The aim of this assignment is to design a cryptocurrency system that works! While Bitcoin is decentralised and depends on a distributed consensus protocol running for all users, in our system a Central Authority (CA) receives transactions from users and is responsible for validating the transactions and adding it to the blockchain. We have given you the skeleton code for creating transactions, users and the cryptosystems required (for signature and hashes), you are required to write the code for the functioning of the CA (both verification and adding to the blockchain).

Here is a list of classes you're given:

- **Transaction.java -** A Transaction class represents a single coin in our currency. The Transaction class has two inner classes, Transaction.Input and Transaction.Output.

  A transaction input consists of the hash of the transaction that contains the corresponding output, the index of this output in that transaction (indices are simply integers starting from 0), and a signature. For the input to be valid, the signature must be on an appropriate digest of the current transaction (see the `getRawDataToSign(int index)` method) with the private key that corresponds to the public key in the output that this input is claiming.

  A transaction output consists of a value and a public key to which it is being paid.

  A transaction consists of a list of its inputs and outputs and a hash of the complete transaction (see the `getRawTx()` method), and contains methods to add and remove an input, add an output, compute digests to sign/hash, add a signature to an input (the computation of signatures is done outside the Transaction class by an entity that knows the appropriate private keys), and compute and store the hash of the transaction once all inputs/outputs/signatures have been added.

- **UTXO.java** - A UTXO class represents an unspent transaction output. A UTXO contains the hash of the transaction from which it originates as well as its index in that transaction. The equals(), hashCode(), and compareTo() methods in UTXO are overridden to provide for equality and comparison between two UTXOs based on their indices and the contents of their txHash arrays instead of their locations in memory.

- **UTXOPool.java -** This class represents the current set of outstanding UTXOs and contains a map from each UTXO to its corresponding transaction output. This class contains constructors to create a new empty UTXOPool or a defensive copy of a given UTXOPool, and methods to add and remove UTXOs from the pool, get the output corresponding to a given UTXO,check if a UTXO is in the pool, and get a list of all UTXOs in the pool. (Recall that HashMap looks up keys by using their hashCode and equals methods.)

- **TxHandler.java -** This is the file that you need to work on, the core logic of the CA. You need to implement the following API:

```
public class TxHandler {
```

```
    /* Creates a public ledger whose current UTXOPool (collection of
     * unspent transaction outputs) is utxoPool. This should make a
     * defensive copy of utxoPool by using the UTXOPool(UTXOPool uPool)
     * constructor.
     */
     public TxHandler(UTXOPool utxoPool);

    /* Returns the current UTXO pool.
     * If no outstanding UTXOs, returns an empty (non-null) UTXOPool
     * object.
     */
     public UTXOPool getUTXOPool();

    /* Returns true if
     * (1) all outputs claimed by tx are in the current UTXO pool,
     * (2) the signatures on each input of tx are valid,
     * (3) no UTXO is claimed multiple times by tx,
     * (4) all of tx's output values are non-negative, and
     * (5) the sum of tx's input values is greater than or equal to the
           sum of its output values; and false otherwise.
     */
     public boolean isValidTx(Transaction tx);

    /* Handles each epoch by receiving an unordered array of proposed
     * transactions, checking each transaction for correctness,
     * returning a mutually valid array of accepted transactions,
     * and updating the current UTXO pool as appropriate.
     */
     public Transaction[] handleTxs(Transaction[] possibleTxs);
}
```

Your implementation of handleTxs() should return a mutually valid transaction set of maximal size (one that can't be enlarged simply by adding more transactions). It need not compute a set of maximum size (one for which there is no larger mutually valid transaction set). Based on the transactions it has chosen to accept, handleTxs should also update its internal UTXOPool to reflect the current set of unspent transaction outputs, so that future calls to handleTxs/isValidTx are able to correctly process/validate transactions that claim outputs from transactions that were accepted in a previous call to handleTxs.

- **TestTxHandler.java** - This is the driver code for the TxHandler, that reads some input files (given to you in the *files/* folder) and tests the validity of transactions using TxHandler class.

Steps to run:

If using Eclipse, you need to right click and run the file TestTxHandler.java. It will print out the test cases that your code passed/failed.

Part B - Blockchain

In this part you will implement a node that's part of a block-chain. Specifically, your code will receive incoming transactions and blocks and maintain an updated block chain.

Files provided:

- **Block.java -** Stores the block data structure.
- **BlockHandler.java -** Uses BlockChain.java to process a newly received block, create a new block, or process a newly received transaction.
- **ByteArrayWrapper.java -** A utility file which creates a wrapper for byte arrays such that it could be used as a key in hash functions.
- **TransactionPool.java -** Implements a pool of transactions, required when creating a new block.
- **DropboxTestBlockChain.java -** A tester class for Blockchain, reads some files from the *files/* folder and tests the validity of blockchain as created.

File to be modified:
BlockChain.java

```
/* BlockChain should maintain enough blocks in memory to implement the
functions below, but should not keep the entire block chain in memory */

public class BlockChain {

public static final int CUT_OFF_AGE = 10;

    // Code provided to you
    private class BlockNode {}

   /* Create an empty block chain with just a genesis block.
    * Assume genesis block is a valid block containing nothing but a
    * coinbase transaction.
    */
    public BlockChain(Block genesisBlock) {
        // IMPLEMENT THIS
    }

   /* Get the maximum height block
    */
    public Block getMaxHeightBlock() {
        // IMPLEMENT THIS
    }

   /* Get the UTXOPool for mining a new block on top of
```

```
     * max height block
     */
     public UTXOPool getMaxHeightUTXOPool() {
         // IMPLEMENT THIS
     }

    /* Get the transaction pool to mine a new block
     */
     public TransactionPool getTransactionPool() {
         // IMPLEMENT THIS
     }

    /* Add a block to block chain if it is valid.
     * For validity, all transactions should be valid
     * and block should be at height > (maxHeight - CUT_OFF_AGE).
     * For example, you can try creating a new block over genesis block
     * (block height 2) if blockChain height is <= CUT_OFF_AGE + 1.
     * As soon as height > CUT_OFF_AGE + 1,you cannot create a new block at
     * height 2. Return true of block is successfully added
     */
     public boolean addBlock(Block b) {
         // IMPLEMENT THIS
     }

    /* Add a transaction in transaction pool
     */
     public void addTransaction(Transaction tx) {
          // IMPLEMENT THIS
     }
}
```

Steps to run:
If using Eclipse, you need to right click and run the file DropboxTestBlockChain.java. It will print out the test cases that your code passed/failed.

Assumptions and Guidelines:
● A new genesis block would not be mined. If you receive a block which claims to be a genesis block (parent is a null hash) in the addBlock(Block b) function, you can return false.
● If there are multiple blocks at the same height, return the oldest block in getMaxHeightBlock() function.
● Assume for simplicity, a coinbase transaction of a block is available to be used in the next block mined on top of it. (This is contrary to the actual Bitcoin protocol when there is a gap of 100 blocks only after which the coinbase transaction can be used).
● Maintain only one global Transaction Pool for the block chain and keep adding transactions to it on receiving transactions and keep removing transactions from it if a new block is received or created. This might cause some transactions to be lost. For example, a block is received on Chain A including transaction Tx1. We remove Tx1 from the transaction pool. Now suppose chain B offshoots chain A. Do not put Tx1 back in the pool, although ideally it should be put back in. This is to simplify your work as

well as our work in testing. (Miners are not responsible for including transactions in the blocks. If a transaction is lost, it is the responsibility of the transaction owner to re-broadcast it in the network).

- The coinbase value is kept as constant in our entire block chain (= 25 bitcoins) whereas we know it changes every four years.
- When checking for validity of a newly received block, just checking if the transactions form a valid set is enough. The set need not be a maximum possible set of transactions. Also, you should not check for hash of the block to contain specific zeros (no proof of work here).

Some useful (just for fun) links:

Link to a book on cryptocurrency:

1. https://d28rh4a8wq0iu5.cloudfront.net/bitcointech/readings/princeton_bitcoin_book.pdf

A TED talk talking about the potential uses and effect of bitcoin:

1. http://www.ted.com/talks/don_tapscott_how_the_blockchain_is_changing_money_and_business

Creating your own cryptocurrency using Ethereum:

1. https://www.ethereum.org/token

Creating your first digital ‘contract’ using Ethereum:

1. https://www.ethereum.org/greeter

Bitcoin miners online:

1. https://blockchain.info/nodes-globe?series=topBlockRelay