Task

Contact

📧 khagan.karimov@utah.edu

🐦 twitter.com/KhaganKarimov

Student: Khagan Khan Karimov

Course: CS 6490-001 Network Security

Instructor: Prof. Dr. Kasera

▼ Task Content:

In ScroogeCoin, the central authority Scrooge receives transactions from users. You will implement the logic used by Scrooge to process transactions and produce the ledger. Scrooge organizes transactions into time periods or blocks. In each block, Scrooge will receive a list of transactions, validate the transactions he receives, and publish a list of validated transactions.

Note that a transaction can reference another in the same block. Also, among the transactions received by Scrooge in a single block, more than one transaction may spend the same output. This would, of course, be a double-spend, and hence invalid. This means that transactions cannot be validated in isolation; it is a tricky problem to choose a subset of transactions that are valid together.

You are provided with a Transaction class that represents a ScroogeCoin transaction and has inner classes Transaction.Output and Transaction.Input. A transaction output consists of a value and a public key to which it is being paid. 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 digital 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 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.

You will also be provided with a UTXO class that 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.

Furthermore, you are provided with a UTXOPool class that represents the current set ofoutstanding UTXOs and contains a map from each UTXO to its corresponding transaction output. This class includes 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.

Finally, you are provided with an rsa.jar file for using the RSAKey class. The public key in a transaction output and the private key used to create signatures in transaction inputs are both represented by an RSAKey, and an RSAKeyPair is a public/private key pair. APIs for RSAKey and RSAKeyPair are provided in the README file. You will be responsible for completing the file called TxHandler.java. Your implementation of *handleTxs* should return a mutually valid transaction set of maximal size (one that cannot 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*.

Turn in your code (in plain text) along with the output files. While you can use any computer to work on your programs, make sure that these run on the CADE lab machines. We cannot grade any programs that do not run on the CADE lab machines. VERY IMPORTANT: Please provide a short readme file with any instructions to run your programs on the CADE machines. Your code must be in Java!