CSE 664: PROJECT REPORT

BUILDING THE BASIC ARCHITECTURE OF A BLOCKCHAIN

- NAGENDRA SATISH KAMATH (Person#: 5024 7661)

1. Project Overview:
    a. Target Functionality: To demonstrate the basic structure of a blockchain, the cryptographic techniques used for secure communication among the users, methods used for securing the data in the blockchain and performing checks to detect any alteration in the data.
    b. Security Model and Objectives:
        i. Desired Objectives:
            - Authenticating users that make transactions
            - Validate transactions so that currency/token is neither created nor destroyed i.e. the sum of deposits and withdrawals is 0 and also prevent overdrafts
            - Each of the transactions are valid updates to the system state
            - Validating blocks by checking the block hash, checking that the block identifier increments the previous block's identifier by 1 and it accurately references the previous block
            - Validating the entire chain by checking the links between the blocks up to the genesis block (first block in the chain) as well as checking if the hash of the entire chain is unchanged
        ii. Information to be protected from:
            - Adversaries that try to insert transactions in their own favor into one of the blocks in the blockchain
            - Adversaries that try to interfere between the two legitimate communicating parties to change the underlying data (e.g. change the transaction amount, change the receiver of the transaction, etc.)
            - Adversaries that make use of GPU based bitcoin mining machines for attacks
    c. Description of Cryptographic Design:
        i. For a new user, their username and their initial balance is taken as input (In real world applications, the initial balance would the amount of cryptocurrency bought by the user).
        ii. A RSA Public Key - Private Key pair is generated for the user, using Crypto.PublicKey.RSA library which uses the Fortuna family of Pseudo Random Generators that make use of os.urandom() of the Linux kernel and random analog inputs. The public exponent used is 65537 and the key size is 2048 bits. [1]
        iii. The user exchanges usernames and public keys from other legitimate user(s). Sockets have been used for this purpose.
        iv. The genesis block is created and added to the blockchain. It consists of the initial balances of the users, no parent block hash and block hash of the genesis block itself. It also has place to store the chain hash.
        v. For making a transaction, the sender inputs the amount to be sent. The amount is checked against the sender's balance to prevent overdrafts.
        vi. The transaction data is created, consisting of sender's username, receiver's username and the amount. This data is encrypted using the receiver's public key and RSA based encryption scheme Crypto.Cipher.PKCS1_OAEP [2]
        vii. Now, the encrypted transaction data is signed using the sender's private key using the RSA based signature algorithm Crypto.Signature.pkcs1_15 [3] and SHA3-256 from the Crypto.Hash.SHA3_256 [4].
        viii. Then the encrypted transaction data and the signature are sent to the receiver. The receiver verifies the signature using the sender's public key, and the hash of the encrypted transaction.

ix. If the signature is valid, the transaction data is decrypted using the receiver's private key and an acknowledgement is sent to the sender. The amount sent is then deducted from the sender's account and added to the receiver's account. The transaction data is added to the current transaction block.
x. If the number of transactions in the current transaction block is equal to the block size limit, the block hash is calculated and linked to the last block in the chain by storing the hash of the previous block in the current block.
xi. The chain hash is calculated using the hash of the combined hash of all the blocks in the block chain and is stored in the genesis block. It is updated after every new block is added to the chain.
xii. The block hash, chain hash and the links can be checked using the options provided to the user. For checking the links of the chain, we start from the genesis block and see if the hash of the block is stored as the parent hash in the next block. We also check the block hash at the same time.

d. All security objectives of part (b) are met.

2. Outcome of the Project:
   a. Project Setup:
      i. Programming Language: Python 2.7
      ii. Platform Used: Ubuntu 16.04 LTS
   b. Results of the implementation:
      i. Security Analysis: A 2048-bit key is used for RSA which is secure up to the year 2030 and also used SHA3-256 which is also one of the recommended settings. [5]
      ii. Scalability Issues: The project was not implemented over a distributed network, which is how its real life implementation is.
   c. Description of what was achieved with respect to the goal: The security objectives were met, although implementing this over a network should have been a better demonstration but since it was a single person project, it was not achieved.
   d. Difficulties Encountered: None
   e. How testing was performed:
      i. Tested for different types of inputs
      ii. Functions for comparing hashes are written
      iii. Care is taken to not send any important data as plaintext by sniffing the data transferred over the network.

3. Source Code:

Alice.py:

```
#!/usr/bin/env  python2
#Author: nkamath

#Building the basic architecture of a blockchain
#Current implementation only supports two users
#Alice (User A) side code

from Crypto.Signature import pkcs1_15 as PKCS1_v1_5
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP as RSACipher
from Crypto.Hash import SHA3_256 as sha3
import json, socket

selfInfo = {}  #dictionary that stores self information
info = {}      #dictionary that is used for sharing self information
otherUser = {} #dictionary to store information of other user
transactionBlock = [] #a block of transactions
chain = []
```

```
blockSizeLimit = 3

def hashMe(msg=""):
    # this is a helper function that wraps our hashing algorithm
    if type(msg)!=str:
        msg = json.dumps(msg,sort_keys=True)  # If we don't sort keys, we can't
guarantee repeatability

    return sha3.new(str(msg).encode('utf-8')).hexdigest()


def addUser():
        #function to help add new users to the blockchain
        print 'We will now be adding you as a new user... '
        username, initialBalance = raw_input('Enter Username and Initial Balance
seperated by space: ').strip().split()

        initialBalance = int(initialBalance)

        if initialBalance < 0:
                initialBalance = 0

        selfInfo['username'] = username
        selfInfo['balance'] = str(initialBalance)
        new_key = RSA.generate(2048)
        public_key = new_key.publickey().exportKey("PEM")
        private_key = new_key.exportKey("PEM")
        selfInfo['public_key'] = public_key
        selfInfo['private_key'] = private_key


def shareInfo():
        #function to share user information for future transactions
        #only Alice' side in this code
        # makeConnection(1) #server socket connection

        info['username'] = selfInfo['username']
        info['balance'] = selfInfo['balance']
        info['public_key'] = selfInfo['public_key']

        #send self info as serialized json object
        serialized_data = json.dumps(info) #sending self info as json object
        # print str(len(serialized_data))
        conn.sendall(str(len(serialized_data))) #first send the length :/
        conn.sendall(serialized_data)

        #receive info of other user as serialized json object
        data_size = conn.recv(len(str(len(serialized_data)))) #receive the size of data
first, assuming length to be approx imately the same
        # print str(data_size)
        serialized_data = conn.recv(int(data_size))
        # print serialized data

        global otherUser
        otherUser = json.loads(serialized_data)

        # converting unicode objects in dictionary to normal strings

        # otherUser = {k.encode('utf8'): v.encode('utf8') for k, v in otherUser.items()}

        # print 'end'
```

```
def makeTransaction():
        global otherUser
        global selfInfo
        global transactionBlock
        if int(raw_input('Send (1) or Receive(2) ? ').strip()) == 1:
                #send
                amount = int(raw_input('Enter amount to be sent: ').strip())
                if amount > int(selfInfo['balance']):
                        print 'Cannot allow overdrafts!'
                        return -1

                transaction = {}

                transaction['sender'] = selfInfo['username']
                transaction['receiver']       = otherUser['username']
                transaction['amount'] = amount

                #need to encrypt the transaction using the otherUser's public key
                cipher = RSACipher.new(RSA.importKey(otherUser['public_key']))
                encrypted_transaction = cipher.encrypt(json.dumps(transaction))
                # print type(encrypted_transaction) #just to check whether it is a string
or not

                #now we digitally sign the transaction
                # h = hashMe(transaction) #serialized and hashed
                h = sha3.new(encrypted_transaction)
                signer = PKCS1_v1_5.new(RSA.importKey(selfInfo['private_key']))
                signature = signer.sign(h)

                #attach signature to the data
                # data = encrypted_transaction + '||' + signature

                #send the transaction as json object along with the digital signature
                # data_size = str(len(encrypted_transaction))
                # conn.sendall(data_size)
                conn.sendall(encrypted_transaction)

                # data_size = str(len(signature))
                # conn.sendall(data_size)
                conn.sendall(signature)

                ack = conn.recv(1) #ack
                if ack == 'T':
                        selfInfo['balance'] = str(int(selfInfo['balance']) - int(amount))
                        otherUser['balance'] = str(int(otherUser['balance']) +
int(amount))
                        addToTransactionBlock(transaction)
                        return True


        else:
                #receive
                # data_size = conn.recv(6) #receive the data size
                # print data_size
                encrypted_transaction = conn.recv(256) #receive the data

                signature = conn.recv(256) #receive the data

                # print len(encrypted_transaction)
                # print 'lol'
                # print len(signature)

                # encrypted_transaction, signature = data.split('||')
```

```
                h = sha3.new(encrypted_transaction)
                verifier = PKCS1_v1_5.new(RSA.importKey(otherUser['public_key']))
                try :
                        verifier.verify(h, signature)
                        print "Transaction's signature verified... "
                        #now we decrypt the transaction and load into transactionBlock
                        decipher = RSACipher.new(RSA.importKey(selfInfo['private_key']))
                        serialized_data = decipher.decrypt(encrypted_transaction)
                        transaction = json.loads(serialized_data)
                        addToTransactionBlock(transaction)
                        conn.sendall('T') #ack

                        amount = transaction['amount']
                        selfInfo['balance'] = str(int(selfInfo['balance']) + int(amount))
                        otherUser['balance'] = str(int(otherUser['balance']) -
int(amount))
                        return True

                except ValueError:
                        print "Invalid Signature!"
                        conn.sendall('F') #ack
                        return False

def displayState():
        print selfInfo['username'], selfInfo['balance']
        print otherUser['username'], otherUser['balance']


def genesisBlock():
        global selfInfo
        global otherUser
        global chain

        genesisBlockTxns = {}
        genesisBlockTxns[selfInfo['username']] = selfInfo['balance']
        genesisBlockTxns[otherUser['username']] = otherUser['balance']
        print genesisBlockTxns

        genesisBlockContents = {'blockNumber': 0,'parentHash': None,'txns':
genesisBlockTxns}
        blockHash = hashMe(genesisBlockContents)

        genesisBlock = {'header': blockHash, 'contents': genesisBlockContents,
'chainHash': None}
        # genesisBlockStr = json.dumps(genesisBlock, sort_keys = True)

        chain.append(genesisBlock)

def calculateChainHash():
        global chain

        totalHash = ''
        for i in chain:
                totalHash += i['header']

        chain_hash = sha3.new(totalHash).hexdigest()

        return chain_hash


def addBlock(txns):
        global chain
        parentBlock = chain[-1]
```

```
        parentHash  = parentBlock['header']
        blockNumber = parentBlock['contents']['blockNumber'] + 1

        blockContents = {'blockNumber':blockNumber,'parentHash':parentHash,'txns':txns}
        blockHash = hashMe(blockContents)
        block = {'header':blockHash,'contents':blockContents}

        chain.append(block)

        #now to update genesis block values
        chain[0]['chainHash'] = calculateChainHash()
        # chain[0]['header'] = hashMe(chain[0]['contents'])


def addToTransactionBlock(transaction):
        global transactionBlock
        transactionBlock.append(transaction)

        if len(transactionBlock) == blockSizeLimit:
                addBlock(transactionBlock)
                transactionBlock = []


def checkChainHash():
        global chain

        print 'Checking Chain Hash now... '
        if calculateChainHash() == chain[0]['chainHash']:
                print 'Chain Hash is Correct!'
        else:
                print 'Hash does not match the value in genesis block!!'
                print chain[0]['chainHash']
        print 'checkChainHash Done.'


def checkBlockHash(block):
        # Raise an exception if the hash does not match the block contents
        expectedHash = hashMe( block['contents'] )
        if block['header'] != expectedHash:
                raise Exception('Hash does not match contents of block %s'%
block['contents']['blockNumber'])



def checkBlockValidity(block,parent):
        # We want to check the following conditions:
        # - Block hash is valid for the block contents
        # - Block number increments the parent block number by 1
        # - Accurately references the parent block's hash

        parentNumber = parent['contents']['blockNumber']
        parentHash   = parent['header']
        blockNumber  = block['contents']['blockNumber']

        #checking block hash for checking block integrity
        checkBlockHash(block) #raises error if inaccurate

        #Checkng if block number increments the parent block number by 1
        if blockNumber != (parentNumber+1):
                raise Exception('Block number is incorrect. The set block number is
%s'%blockNumber)

        #Checkingn if it accurately references the parent block's hash
```

```
        if block['contents']['parentHash'] != parentHash:
                raise Exception('Parent hash not accurate at block %s'%blockNumber)


def checkChain(chain):
        # Work through the chain from the genesis block (which gets special treatment),
        # and that the blocks are linked by their hashes.


        try:
                checkBlockHash(chain[0])
                parent = chain[0]
        except Exception as e:
                print 'Error: '+ repr(e)

        ## Checking subsequent blocks: These need to check
        #    - the reference to the parent block's hash
        #    - the validity of the block number

        for block in chain[1:]:
                try:
                        checkBlockValidity(block,parent)
                        parent = block
                except Exception as e:
                        print 'Error: '+ repr(e)
        print 'checkChain Done.'


def displayCurrentTransactionBlock():
        global transactionBlock
        print transactionBlock
        print 'displayCurrentTransactionBlock Done.'


def displayChain():
        global chain
        for i in chain:
                print i
        print '\nLength of chain: ' + str(len(chain))


#main gotta start here


addUser()

#server side - ALice is always the server for our case
HOST = ''    #localhost
PORT = 50007 #The same port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connection established... '

# print str(conn.recv(len("hi")))

shareInfo()
genesisBlock()

main_choice = '-1'
while int(main_choice) != 7:
```

```
        main_choice = raw_input('\nChoose an option:\n1. Make Transaction\n2. Display
State\n3. Display current transaction block\n4. Display Chain\n5. Check Chain Hash\n6.
Check Chain\n7. Quit\n').strip()

        check_choice = int(main_choice)

        if check_choice == 1:
                makeTransaction()
        elif check_choice == 2:
                displayState()
        elif check_choice == 3:
                displayCurrentTransactionBlock()
        elif check_choice == 4:
                displayChain()
        elif check_choice == 5:
                checkChainHash()
        elif check_choice == 6:
                checkChain(chain)


s.close()
```

Bob.py:

```
#!/usr/bin/env  python2
#Author: nkamath

#Building the basic architecture of a blockchain
#Current immplementation only supports two users
#Bob (User b) side code

from Crypto.Signature import pkcs1_15 as PKCS1_v1_5
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP as RSACipher
from Crypto.Hash import SHA3_256 as sha3
import json, socket


selfInfo = {}  #dictionary that stores self information
info = {}      #dictionary that is used for sharing self information
otherUser = {} #dictionary to store information of other user
transactionBlock = [] #a block of transactions
chain = []
blockSizeLimit = 3

def hashMe(msg=""):
    # this is a helper function that wraps our hashing algorithm
    if type(msg)!=str:
        msg = json.dumps(msg,sort_keys=True)  # If we don't sort keys, we can't
guarantee repeatability

    return sha3.new(str(msg).encode('utf-8')).hexdigest()


def addUser():
        #function to help add new users to the blockchain
        print 'We will now be adding you as a new user... '
        username, initialBalance = raw_input('Enter Username and Initial Balance
seperated by space: ').strip().split()

        initialBalance = int(initialBalance)
```

```
        if initialBalance < 0:
                initialBalance = 0

        selfInfo['username'] = username
        selfInfo['balance'] = str(initialBalance)
        new_key = RSA.generate(2048)
        public_key = new_key.publickey().exportKey("PEM")
        private_key = new_key.exportKey("PEM")
        selfInfo['public_key'] = public_key
        selfInfo['private_key'] = private_key


def shareInfo():
        #function to share user information for future transactions
        #only Bob's side in this code
        # makeConnection(-1) #client socket connection

        info['username'] = selfInfo['username']
        info['balance'] = selfInfo['balance']
        info['public_key'] = selfInfo['public_key']
        serialized_data = json.dumps(info) #info as json object

        # print str(len(str(len(serialized data))))

        #receive info of other user as serialized json object
        data_size = s.recv(len(str(len(serialized_data)))) #receive the size of data
first, assuming length to be approx imately the same
        # print str(data size)

        serialized_data = s.recv(int(data_size))
        # print serialized_data

        global otherUser
        otherUser = json.loads(serialized_data)

        # converting unicode objects in dictionary to normal strings
        # otherUser = {k.decode('utf8'): v.decode('utf8') for k, v in otherUser.items()}
        # print otherUser['username']

        #send self info as serialized json object
        serialized_data = json.dumps(info) #sending self info as json object
        s.sendall(str(len(serialized_data))) #first send the length :/
        s.sendall(serialized_data)

        # print 'end'



def makeTransaction():
        global otherUser
        global selfInfo
        global transactionBlock
        if int(raw_input('Send (1) or Receive(2) ? ').strip()) == 1:
                #send
                amount = int(raw_input('Enter amount to be sent: ').strip())
                if amount > int(selfInfo['balance']):
                        print 'Cannot allow overdrafts!'
                        return -1

                transaction = {}

                transaction['sender'] = selfInfo['username']
                transaction['receiver']       = otherUser['username']
```

```
                transaction['amount'] = amount

                #need to encrypt the transaction using the otherUser's public key
                cipher = RSACipher.new(RSA.importKey(otherUser['public_key']))
                encrypted_transaction = cipher.encrypt(json.dumps(transaction))
                # print type(encrypted_transaction) #just to check whether it is a string
or not

                #now we digitally sign the transaction
                # h = hashMe(transaction) #serialized and hashed
                h = sha3.new(encrypted_transaction)
                signer = PKCS1_v1_5.new(RSA.importKey(selfInfo['private_key']))
                signature = signer.sign(h)

                #attach signature to the data
                # data = encrypted_transaction + '||' + signature

                #send the transaction as json object along with the digital signature
                # data_size = str(len(encrypted_transaction))
                # s.sendall(data_size)
                s.sendall(encrypted_transaction)
                s.sendall(signature)


                ack = s.recv(1) #ack
                if ack == 'T':
                        selfInfo['balance'] = str(int(selfInfo['balance']) - int(amount))
                        otherUser['balance'] = str(int(otherUser['balance']) +
int(amount))
                        addToTransactionBlock(transaction)
                        return True

        else:
                #receive
                # data_size = s.recv(6) #receive the data size
                # print data_size
                encrypted_transaction = s.recv(256) #receive the data

                signature = s.recv(256) #receive the data

                # print len(encrypted_transaction)
                # print 'lol'
                # print len(signature)

                # encrypted_transaction, signature = data.split('||')
                h = sha3.new(encrypted_transaction)
                verifier = PKCS1_v1_5.new(RSA.importKey(otherUser['public_key']))
                try :
                        verifier.verify(h, signature)
                        print "Transaction's signature verified... "
                        #now we decrypt the transaction and load into transactionBlock
                        decipher = RSACipher.new(RSA.importKey(selfInfo['private_key']))
                        serialized_data = decipher.decrypt(encrypted_transaction)
                        transaction = json.loads(serialized_data)
                        addToTransactionBlock(transaction)
                        s.sendall('T') #ack

                        amount = transaction['amount']
                        selfInfo['balance'] = str(int(selfInfo['balance']) + int(amount))
                        otherUser['balance'] = str(int(otherUser['balance']) -
int(amount))
                        return True
```

```
                except ValueError:
                        print "Invalid Signature!"
                        s.sendall('F') #ack
                        return False



def displayState():
        print selfInfo['username'], selfInfo['balance']
        print otherUser['username'], otherUser['balance']


def genesisBlock():
        global selfInfo
        global otherUser
        global chain

        genesisBlockTxns = {}
        genesisBlockTxns[selfInfo['username']] = selfInfo['balance']
        genesisBlockTxns[otherUser['username']] = otherUser['balance']
        print genesisBlockTxns

        genesisBlockContents = {'blockNumber': 0,'parentHash': None,'txns':
genesisBlockTxns}
        blockHash = hashMe(genesisBlockContents)

        genesisBlock = {'header': blockHash, 'contents': genesisBlockContents,
'chainHash': None}
        # genesisBlockStr = json.dumps(genesisBlock, sort_keys = True)

        chain.append(genesisBlock)

def calculateChainHash():
        global chain

        totalHash = ''
        for i in chain:
                totalHash += i['header']

        chain_hash = sha3.new(totalHash).hexdigest()

        return chain_hash


def addBlock(txns):
        global chain
        parentBlock = chain[-1]
        parentHash  = parentBlock['header']
        blockNumber = parentBlock['contents']['blockNumber'] + 1

        blockContents = {'blockNumber':blockNumber,'parentHash':parentHash,'txns':txns}
        blockHash = hashMe(blockContents)
        block = {'header':blockHash,'contents':blockContents}

        chain.append(block)

        #now to update genesis block values
        chain[0]['chainHash'] = calculateChainHash()
        # chain[0]['header'] = hashMe(chain[0]['contents'])
        # block['contents']['parentHash'] =


def addToTransactionBlock(transaction):
```

```
        global transactionBlock
        transactionBlock.append(transaction)

        if len(transactionBlock) == blockSizeLimit:
                addBlock(transactionBlock)
                transactionBlock = []


def checkChainHash():
        global chain

        print 'Checking Chain Hash now... '
        if calculateChainHash() == chain[0]['chainHash']:
                print 'Chain Hash is Correct!'
        else:
                print 'Hash does not match the value in genesis block!!'
                print chain[0]['chainHash']
        print 'checkChainHash Done.'


def checkBlockHash(block):
        # Raise an exception if the hash does not match the block contents
        expectedHash = hashMe( block['contents'] )
        if block['header'] != expectedHash:
                raise Exception('Hash does not match contents of block %s'%
block['contents']['blockNumber'])



def checkBlockValidity(block,parent):
        # We want to check the following conditions:
        # - Block hash is valid for the block contents
        # - Block number increments the parent block number by 1
        # - Accurately references the parent block's hash

        parentNumber = parent['contents']['blockNumber']
        parentHash   = parent['header']
        blockNumber  = block['contents']['blockNumber']

        #checking block hash for checking block integrity
        checkBlockHash(block) #raises error if inaccurate

        #Checkng if block number increments the parent block number by 1
        if blockNumber != (parentNumber+1):
                raise Exception('Block number is incorrect. The set block number is
%s'%blockNumber)

        #Checkingn if it accurately references the parent block's hash
        if block['contents']['parentHash'] != parentHash:
                raise Exception('Parent hash not accurate at block %s'%blockNumber)


def checkChain(chain):
        # Work through the chain from the genesis block (which gets special treatment),
        # and that the blocks are linked by their hashes.

        try:
                checkBlockHash(chain[0])
                parent = chain[0]
        except Exception as e:
                print 'Error: '+ repr(e)

        ## Checking subsequent blocks: These need to check
```

```
        #    - the reference to the parent block's hash
        #    - the validity of the block number

        for block in chain[1:]:
                try:
                        checkBlockValidity(block,parent)
                        parent = block
                except Exception as e:
                        print 'Error: '+ repr(e)

        print 'checkChain Done.'


def displayCurrentTransactionBlock():
        global transactionBlock
        print transactionBlock
        print 'displayCurrentTransactionBlock Done.'


def displayChain():
        global chain
        for i in chain:
                print i
        print '\nLength of chain: ' + str(len(chain))

#main gotta start here

addUser()

#client side - Bob is always the client for our case

HOST = ''    #localhost
PORT = 50007 #port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))

shareInfo()
genesisBlock()

main_choice = '-1'
while int(main_choice) != 7:
        main_choice = raw_input('\nChoose an option:\n1. Make Transaction\n2. Display
State\n3. Display current transaction block\n4. Display Chain\n5. Check Chain Hash\n6.
Check Chain\n7. Quit\n').strip()

        check_choice = int(main_choice)

        if check_choice == 1:
                makeTransaction()
        elif check_choice == 2:
                displayState()
        elif check_choice == 3:
                displayCurrentTransactionBlock()
        elif check_choice == 4:
                displayChain()
        elif check_choice == 5:
                checkChainHash()
        elif check_choice == 6:
                checkChain(chain)

s.close()
```

4. Description of data on which the project was run on: No external data was used.

References:

[1] http://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html

[2] http://pycryptodome.readthedocs.io/en/latest/src/cipher/oaep.html

[3] http://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html

[4] http://pycryptodome.readthedocs.io/en/latest/src/hash/sha3_256.html

[5] https://www.keylength.com/en/4/