Implementing Cryptographic Primitives for BlockChain

Cryptography CS 411 & CS 507 Term Project for Fall 2019

E. Savaş
Computer Science & Engineering
Sabancı University
İstanbul

Abstract

You are required to develop essential building blocks of cryptocurrency using block chains.

1 Introduction

The project has three phases:

- Developing software for digital signature and blockchain transactions
- Developing software for proof of work
- Developing software for other building blocks and integration

More information about the phases are given in the subsequent sections.

2 Phase I: Developing software for digital signature and Blockchain Transactions

In this phase of the project, you are required to upload two files: "DS.py" and "Tx.py". We will be able to test your codes using "PhaseI_Test.py'. If your software cannot be tested by "PhaseI_Test.py' as it is, you will get no credit.

2.1 Developing software for digital signature

Here, you will develop a Python module "DS.py" that includes functions for signing given any message and verifying the signature. For digital signature (DS) you will use an algorithm, which consists of four functions as follows:

- **Public parameter generation:** Two prime numbers p and q are generated with $q|p-1$, where q and p are 224-bit and 2048-bit integers, respectively. The generator g generates a subgroup of $\mathbb{Z}_p^*$ with q elements. Naturally, $g^q \equiv 1 \bmod p$. Note that in your system q, p, and g are *public parameters* shared by all users, who have different secret/public key pairs. Refer to the slide (with title "DSA Setup" in chapter 10 for an efficient method for parameter generation).

- **Key generation:** A user picks a random secret key $0 < \alpha < q-1$ and computes the public key $\beta = g^{\alpha} \bmod p$.
- **Signature generation:** Let m be an arbitrary length message. The signature is computed as follows:
  1. $h = \text{SHA3\_256}(m) \pmod q$
  2. $k \leftarrow \mathbb{Z}_q$, (i.e., k is a random integer in $[1, q-2]$).
  3. $r = (g^k \pmod p) \pmod q$
  4. $s = \alpha \cdot r - k \cdot h \pmod q$
  5. The signature for m is the tuple (s, r).
- **Signature verification:** Let m be a message and the tuple (s, r) is a signature for m. The verificiation proceeds as follows:
  - $h = \text{SHA3\_256}(m) \pmod q$
  - $v = h^{-1} \pmod q$
  - $z_1 = s \cdot v \pmod q$
  - $z_2 = r \cdot v \pmod q$
  - $u = (g^{-z_1}\beta^{z_2} \pmod p) \pmod q$
  - Accept the signature only if $u = r$
  - Reject it otherwise.

Note that the signature generation and verification of this DS are different than those discussed in the lecture.

You are required to develop Python software that implements those four functions; namely Setup for public parameter generation, Key Generation, Signature Generation and Signature Verification. You are required to test your software using the test routines in "PhaseI_Test.py" provided in the assignment package.

In this part of "PhaseI_Test.py", there are four basic test functions:

1. checkDSparams(q, p, g) takes your public parameters (q, p, g) and checks if they are correct. It returns 0 if they are. Otherwise, it returns a code that indicates the problem.
2. CheckKeys(q, p, g, α, β) takes your public parameters (q, p, g) and key pair (α, β) and checks if the key pair is correct. It returns 0 if they are; otherwise it returns -1.
3. CheckSignature(q, p, g, α, β) takes your public parameters (q, p, g), key pair (α, β) and generates a signature for a random message and verifies the signature. It returns 0 if the signatures verifies; otherwise it returns -1.
4. CheckTestSignature() reads the file "TestSet.txt" (provided in the assignment package), which contains public parameters, a public key, and 10 randomly chosen messages and their signatures. The test code reads them and runs signature verification function. The test code returns 0 if all signatures verify; otherwise it returns -1.

2.2 Generating Random Transactions

Here, you will develop a Python module "Tx.py" that includes functions for generating a random Blockchain transaction. A transaction contains information of a payment (transaction) from the *payer* to the *payee* and is in the following format:

```
*** Bitcoin transaction ***
Serial number:
Payer public key (beta):
Payee public key (beta):
Amount:
Signature (s):
Signature (r):
```

Explanations of these fields are as follows
`Serial Number:` is a uniformly randomly generated 128-bit integer
`Payer public key (beta):` is the public key of the person making the payment
`Payee public key (beta):` is the public key of the person receiving the payment
`Amount:` is the amount in *Satoshi* being transferred in range [1, 1000000]
`Signature (s):` Signature (s part) of the transaction by Payer
`Signature (r):` Signature (r part) of the transaction by Payer

Note that the payer and payee are identified by their public keys in the transaction, which is signed by the private key of the payer. Therefore, the transaction can be verified by the public key of the payer.

In this part of "PhaseI_Test.py", there are two test functions:

1. CheckTransaction(q, p, g) creates a random transaction and verifies its signature. For this, you will develop a function named "gen_random_tx(q, p, g)" in "Tx.py".
2. CheckBlockofTransactions() reads sample transactions given in file "transactions.txt" (also provided in the assignment package) and verifies the signatures of all transactions using your verification function in "DS.py"; namely "DS.SignVer".

3 Phase II: Developing software for transaction and implementing proof of work

In this phase of the project, you are required to generate a block of random transactions and Proof-of-Work for the block. The details are given in the following subsections. For further information about bitcoin and blockchain, please refer to "bitcoin.pdf" provided in the assignment package.

3.1 Generating a Block of Transacations

In this part of the project, you are required to generate random transactions and write those transactions into the file "transactions.txt". For this, you need to develop a function with the following interface

$$\text{GEN\_RANDOM\_TXBLOCK}(q, p, g, \text{TxCnt}, \text{filename}),$$

and add it to the file "Tx.py", which you submitted in the first phase of the project. Here, "TxCnt" is the number of transactions in the file named "filename" (in our case "filename" = "transactions.txt"). Also, make sure that "TxCnt" is always a power of two. In the assignment package a sample block of transactions is provided in "transactions_sample.txt".

We will test your transactions in "transactions.txt" using CheckBlock() function in the test code "PhaseII_Test.py" (see Test 1). For this part of Phase II, you are required to submit the files "pubparams.txt" and the new "Tx.py".

3.2 Implementing Proof-of-Work (PoW)

In some cryptocurrency implementations (e.g., bitcoin), blockchain network members (a.k.a. miners) approve transactions by running the proof-of-work (PoW) algorithm. The PoW algorithm is a *consensus protocol* that determines who will write the next block of the transactions to the blockchain. All miners participate in the protocol as they receive a commission if they win.

A miner reads a block of transactions (from the file "transactions.txt"), computes the root hash of the transactions using the Merkle tree, H_r, with SHA3_256. See the slide 22 in the file "ch7_handout.pdf for Merkle Tree implementation.

Then, s/he appends a random number called *nonce* to H_r to compute a hash value of the special form. In particular, SHA3_256($H_r||nonce$) must start with $4 \cdot$ PoWLen 0 bits; i.e., if you print the hash value with "hexdigest()", the first PoWLen hexadecimal digits must be 0. The hash value with this property is known as *Proof-of-Work* or shortly *PoW*. To find a PoW, the miner tries different nonce values chosen at random. A sample PoW with PoWLen=5 looks as follows:

$$000006a5a2a3cfae49f6cdfdafcb11d0a6995bf6acf9cefedf986aedbac10924.$$

The miner, who comes up with such hash value first, wins the right to add the block to the chain and thus recevies the commission. A sample block is given in file "block_sample.txt" with PoWLen = 5 provided in the assignment package. Note that "block_sample.txt" is almost the same as "transactions_sample.txt" except for the last line

Nonce : 330915583940940209115447795171280728973.

For Test 2, you are required to develop the function

CheckPow($p, q, g,$ PoWLen, TxCnt, filename)

in the file "PoW.py". Here, TxCnt is the number of transactions in the file "filename". The function CheckPow returns an empty string if PoW of the block of transactions in "filename' does not have preceding PoWLen hexadecimal 0s. Otherwise, it returns the value of PoW. In Test 2, for the block of transaction in "block_sample.txt" PoWLen = 5.

In Test 3, you are required to write a function

PoW(PoWLen, $q, p, g,$ TxCnt, filename)

that reads the transactions in filename (i.e., "transactions.txt" generated in Test 1) and computes a PoW for the block. PoWLen must be at least 5. Once your programs finds the PoW for the block, it appends the nonce at the end of the block and writes it into a file with the name "block.txt". A sample block is provided in the file "block_sample.txt" in the assignment package.

You are required to test your codes with the test functions in “PhaseII_Test.py” before submission as we will use it to test your codes. If your codes will not pass the tests in “PhaseII_Test.py”, you will get no credit in this phase of the project.

Finally, as generating PoW takes quite some time, you are recommended to test your codes with smaller PoWLen first such as PoWLen = 3 to make sure your code is working. Then you try larger PoWLen.

3.3 Bonus I - Competition

The group that will submit a block with the longest PoW will recieve an extra 10% in this Phase. Submit your block in the file “block_[yournames].txt”. To qualify for the competition, PowLen must be at least 7. In case of tie, the earliest submission wins. Make sure that “block.txt’ is testable by the “Test 2’ in “PhaseII_Test.py”. Send an e-mail to “eahsen@sabanciuniv.edu” and “erkay.savas@sabanciuniv.edu” to notify us that you are in the competition.

4 Phase III: ECDSA Integration and Blockchain Generation

In this phase of the project you will work on two parts. In the first part, you will integrate elliptic curve digital signature algortihm (ECDSA) to your implementation. In the second part, you will create a *blockchain* by linking blocks using PoW. The details are given in the subsequent sections.

4.1 ECDSA Integration

You will implement the elliptic curve version of the signature scheme in Phase I. As the signature scheme is now different transactions will be different as well; e.g., there will be two lines each for payer and payee in a transaction, since public keys are elliptic curve points now. For sample transactions, see “transactions.txt” provided in the assignment package.

For the implementation of ECDSA, you will use “ecpy” module, which should be installed first (running “pip install ecpy”). See the code segments in the file “ECC_Sample.py” in the assignment package as to how to perform elliptic curve arithmetic and ECDSA operations.

In this part of the project, you will upload only one file: “ECDSA.py” that implements the signature scheme. Your implementation must pass the first three tests in “PhaseIII_Test.py” file. The tests are:

- **Test I:** generates a random key pair, signs a message and verifies the signature.
- **Test II:** given a message, a public key and a signature, verifies the signature.
- **Test III:** generates a random block of transactions, signs every transaction, and verifies the transaction signatures.

4.2 Blockchain Generation

Now it is time to generate a blockchain by linking a block to another block. In order to form a chain, every block in the blockchain contains an additional field called “Previous PoW” that is actually the hash of the previous block in the chain. Note that this hash value is PoW of the previous block and its PoWLen most significant digits are 0. Five files that contain one such block each forming a blockchain of length 5 are given in the assignment package (see “Block0.txt”, “Block1.txt”,“Block2.txt”, “Block3.txt”, and “Block4.txt”).

In this part of the project, you will develop the function

```
AddBlock2Chain(PoWLen, TxCnt, PrevBlock, block_candidate)
```

in “ChainGen.py” that takes the last block in the chain (i.e., previous block, `PrevBlock`) and a block of transactions (`block_candidate`), and returns a new block and PoW of the previous block. The new block contains the same transactions in `block_candidate` with two additional lines at the end: i) Previous PoW and ii) Nonce. The nonce is a random integer such that the hash of the block with the nonce contains PoWLen hexadecimal 0s in front. PoW of a block is computed as the hash of the following:

```
H_r + PrevPoW + nonce.to_bytes((nonce.bit_length()+7)//8, byteorder = 'big')
```

If the first block of the chain is generated, the function `AddBlock2Chain` is called with `PrevBlock = ""` and it is assumed that Previous PoW is `00000000000000000000`.

In this part of the project, you will upload only one file: “ChainGen.py” that implements the `AddBlock2Chain` function. Your implementation should pass Test IV in ‘PhaseIII_Test.py” file.

Test IV performs the following operations:

1. generates a random block that contains `TxCnt` transactions and writes them in a file.
2. reads the transactions from the file, calls `AddBlock2Chain` function, generates the new block, writes it to a new file. It also checks if the previous PoW is good.
3. forms a block chain of length `ChainLen` by repeating Steps 1 and 2.
4. writes the resulting blocks in files with names “Block0.txt”, “Block1.txt”,“Block2.txt”, ...
5. reads all blocks in the chain and performs two checks:
   - whether a block contains the hash of the previous block.
   - whether its PoW is correct.

4.3 Bonus II - Competition

The challenge is to generate the longest blockchain with PowLen = 9. In case of a tie, the earliest submission wins. Make sure that your blocks are testable by the “Test IV” in “PhaseIII_Test.py”. Submit a file named “chain_[yournames].txt” that only contains the length of your blockchain. **Do NOT upload your blocks to SUCourse.**

5 Appendix I: Timeline & Deliverables & Weight & Policies etc.

Project Phases	**Deliverables**	**Due Date**	**Weight**
Project announcement		29/11/2019	
First Phase	Files: DS.py and Tx.py	6/12/2019	30%
Second Phase	Files: DS.py, Tx.py and PoW.py pubparams.txt	13/12/2019	30%
Bonus - I	block_[yournames].txt	13/12/2019	10%
Third Phase	Files ECDSA.py, ChainGen.py,	22/12/2019	40%
Bonus - II	chain_[yournames].txt	22/12/2019	10%

5.1 Policies

- You may work in groups of two.
- Submit all deliverables in the zip file "cs411_507_tp3_yourname.zip".
- You may be asked to demonstrate a project phase to a TA or the instructor.
- In every phase, we will provide you with a validation software in Python language that can be used to check your implementation for correctness. We will also use it to check your implementation. If your implementation in a project phase fails to pass the validation, you will get no credit for that phase.
- Your codes will be checked for their similarity to other students' codes; and if the similarity score exceeds a certain threshold you will not get any credit for your work.