Implementing Cryptographic Primitives for BlockChain

Cryptography CS 411& CS 507 Term Project for Fall 2018

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
- 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

In this phase of the project, you will develop software for signing given any message. For digital signature (DS) you will us 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 $0 < \alpha < q$ 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. $k \leftarrow \mathbb{Z}_q$, (i.e., k is a random integer in $[0, q-1]$).

2. $r = g^k \pmod p$
3. $h = \text{SHA3\_256}(m||r)$
4. $s = \alpha \cdot h + k \pmod q$
5. The signature for m is the tuple (s, h).

- **Signature verification:** Let m be a message and the tuple (s, h) is a signature for m. The verificiation proceeds as follows:
  - $v = g^s \beta^{-h} \pmod p$
  - $\tilde{h} = \text{SHA3\_256}(m||v)$
  - Accept the signature only if $h = \tilde{h} \bmod q$
  - 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 "DS_Test.py" provided in the assignment package.

In "DS_Test.py", there are four basic test functions:

1. checkDSparams(q, p, g) takes your public parameters (q, p, g) and check 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 check 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.

In this phase of the project, you are required to upload only a file named "DS.py" with sufficient comments. We will be able to test your code using "DS_Test.py'. If your software cannot be tested by "DS_Test.py' as it is, you will get no credit.

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.pptx provided in the assignment package.

3.1 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 (h):
```

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 (h):` Signature (h 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. For the signature you are required to use the set of public parameters in file "pubparams.txt" provided in the assignment package. A sample block of random transactions is given in file "transactions_sample.txt" also provided in the assignment package.

In this part of the project, you are required to generate random transactions and write those transactions into a file named "transactions.txt". For this, you will develop a function named "gen_random_tx(q, p, g)". You have to test your transactions in "transactions.txt' using CheckBlock() function in Test code "PhaseII_Test.py' before submission.'

As a deliverable, you are required to submit a file with the name "Tx.py" that should include the function

"gen_random_tx(q, p, g)"
that will take the public parameters and output a random transaction in the format defined above.

3.2 Implementing Proof-of-Work (PoW)

In cryptocurrency systems, 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 commision if they win.

A miner reads a block of transactions, adds a random number called *nonce* at the end and tries to compute a hash value of the special form. In particular, the hash value must start with x 0 bits; i.e., if you print the hash value with "hexdigest()", the first PoWLen = $x/4$ hexadecimal digits must be 0 (pick x a multiple of 4). The hash value with this property is known as *Proof-of-Work* or shortly *PoW*. To find PoW, the miner tries different nonce values chosen at random.

The miner, who comes up with such hash value first, wins the right to add the block to the chain and thus recevies the commision. A sample block is given in file "block.txt" with $x = 20$

provided in the assignment package.

You are required to write a function (with the name "PoW") that reads the transactions in the file "transactions.txt" 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 strongly recemmended 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.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 II" in "PhaseII_Test.py".

4 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 repeat the Phase II of the project using ECDSA instead of DSA. As the signature scheme is now different transactions will be different as well; e.g., as public keys are elliptic curve points now, there will be two lines each for payer and payee in a transaction. 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" or "pip3 install ecpy"). See the code segments in the file "ECDSA.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 two files: "TxECDSA.py" and "PoWECDSA.py". Make sure your implementations should pass the tests in "Test I" and "Test II" parts of the test file "PhaseIII_Test.py' in the assignment package.

Test I first generates 32 transactions, write them in the file "transacstions.txt", checks the signatures of all transactions in the file. Test II first reads the transactions in "transactions.txt", finds PoW for them and write all transactions in a different file "block.txt". Then it checks PoW of the block in the file.

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 Hash" 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. Four files that contain one such block eachs forming a blockchain of length 4 are given in the assignment package (see "Block0.txt", "Block1.txt","Block2.txt", "Block3.txt").

In this part of the project, you will develop the function `AddBlock2Chain(PoWLen, PrevBlock, NewBlock)` in "ChaingGen.py" that takes the last block in the chain (i.e., "PrevBlock") , computes its hash and add a field named "PrevHash" in "NewBlock", computes PoW for it. It returns a block that contains "PrevHash" and "Nonce" fields in addition to the transactions in NewBlock. If NewBlock is the first block its PrevHash field is set to `b'0'` (For this, call `AddBlock2Chain` function with its second parameter (PrevBlock) is set to 0).

Make sure your code passes "Test III" in PhaseIII_Test.py. Test III first generates blocks of transactions, forms a blockchain from them and writes the resulting blocks in files with names "Block0.txt", "Block1.txt","Block2.txt", "Block3.txt' ..... Then, it reads all blocks 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 = 7. In case of tie, the earliest submission wins. Make sure that your blocks are testable by the "Test III" in "PhaseIII_Test.py". Submit a file named "BonusII.txt" that only contains the length of your blockchain. **Do NOT upload your blocks to SUCourse.**

4.4 Bonus III

Implement the elliptic curve version of the signature algorithm given in Section 2 (Phase I). To test your code, use "ECDSA_Test.py" in the assignment package and make changes in the indicated lines.

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

5.1 Policies

- You may work in groups of two.
- 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.

Project Phases	**Deliverables**	**Due Date**	**Weight**
Project announcement		07/12/2018	
First Phase	Source code (DS.py)	14/12/2018	25%
Second Phase	Source codes (Tx.py and PoW.py)	21/12/2018	40%
Bonus - I	block.txt	21/12/2018	10%
Third Phase	Source codes (TxECDSA.py, PoWECDSA.py, ChainGen.py)	28/12/2018	35%
Bonus - II	Bonus.txt	28/12/2018	10%
Bonus - III	Source code (ECDSA_Test.py)	28/12/2018	5%