Implementing Cryptocurrency with Block Chains

Cryptography CS 411& CS 507 Term Project for Fall 2017

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 proof of work
- Developing software for digital signature
- Developing software for other buiding blocks and integration

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

2 Phase I: Developing software for proof of work

In cryptocurrency systems, miners approve transactions by running the proof-of-work (PoW) algorithm. A transaction contains information of a payment from the *payer* to the *payee* and is in the following format:

```
*** Bitcoin transaction ***
Serial number:
Payer:
Payee:
Amount:
Previous hash in the chain:
Nonce:
Proof of Work:
```

Explanations of these fields are as follows
`Serial Number:` is a uniformly randomly generated 128-bit integer;
`Payer:` is the identity of the person making the payment;
`Payee:` is the identity of the person receiving the payment;

`Amount:` is the amount in *Satoshi* being transferred;
`Previous hash in the chain:` is the hash of the last transaction in the chain
`Nonce:` is a uniformly randomly generated 128-bit integer, which miners change to find the PoW for transactions.
`Proof of Work:` is the hash of transactions whose first digits of certain length are all 0. This is written in hexadecimal format.

The following is an example of a block chain generated for three transactions:

```
*** Bitcoin transaction ***
Serial number: 212263538645046771822600700406703677780
Payer: Erkay Savas
Payee: 2KQMHBX7LR
Amount: 241 Satoshi
Previous hash in the chain: First transaction
Nonce: 295735282256603556348499786196010373709
Proof of Work: 00000042e244ebcdf74f456c35f674d18d014b1317c335beb717fb6af3012403
*** Bitcoin transaction ***
Serial number: 15406649440213522152658022974697492647
Payer: Erkay Savas
Payee: JJCHLEOS8S
Amount: 353 Satoshi
Previous hash in the chain: 00000042e244ebcdf74f456c35f674d18d014b1317c335beb717fb6af3012403
Nonce: 229011888470430239833581862495372060758
Proof of Work: 00000039f0eb03ca50d6d7399717962e47f1189b70fc5b4abfea1c6df4436100
*** Bitcoin transaction ***
Serial number: 134664886688001698414829205535324634147
Payer: Erkay Savas
Payee: 8O41AGH4QM
Amount: 711 Satoshi
Previous hash in the chain: 00000039f0eb03ca50d6d7399717962e47f1189b70fc5b4abfea1c6df4436100
Nonce: 174041738328075334247840790160731353141
Proof of Work: 000000b48355d57cf79a79801ec96cbd83716b144a0d2896a9078a14b39fa202
```

As can be observed from this example, all PoWs start with six 0s. This can be achived by miners generating different `Nonce` values until one hash satisfying this condition is found. PoW is the hash of the first seven lines of a transaction excluding the last line. In the example we use `SHA3` with 256-bit output as the cryptographic hash function in python module `hashlib`.

In this phase of the project, you will generate random transactions following the format described above. You are required to use `SHA3` with 256-bit output as the cryptographic hash function. For the sake of simplicity, you need to provide a PoW for a single transcation as in the example in the file "LongestChain.txt" attached to the assignment package. You need to generate at least ten transactions and write them to a file.

You are required to submit your python source code(s) that you used to generate transactions and their PoWs along with a file that contains at least ten transactions. The transactions in your file should be in the format as the example transactions in the file "LongestChain.txt". Note that we will use the python code "ValidateChain.py" in the attachment to validate your transactions. Before you submit your work, you are advised to use it to check if your transactions are good.

For more information about cryptocurrencies see the attached document "bitcoin.pptx".

3 Phase II: Developing software for digital signature

In this phase of the project, you will develop software for signing a bitcoin transaction. For digital signature you will use a digital signature algorithm (DSA) which consists of four functions as follows:

- **Parameter generation:** Two prime numbers p and q are generated with $q|p-1$, where q and p are 256-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 domain parameters shared by all users, who have different secret/public key pairs. Refer to the slide number 13 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:
  - $h = \text{SHA3\_256}(m)$
  - $h = h \bmod q$
  - $r = (g^k \bmod p) \bmod q$, where $0 < k < q$ is random integer.
  - $s = \alpha \cdot r + k \cdot h \bmod q$
  - The signature for m is the tuple (r, s).
- **Signature verification:** Let m be a message and the tuple (r, s) is a signature for m. The verificiation proceeds as follows:
  - $h = \text{SHA3\_256}(m)$
  - $h = h \bmod q$
  - $v = h^{-1} \bmod q$
  - $z_1 = s \cdot v \bmod q$
  - $z_2 = (q - r) \cdot v \bmod q$
  - $u = g^{z_1} \cdot \beta^{z_2} \bmod p$
  - Accept the signature only if $r = u \bmod q$
  - Reject it otherwise.

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

You will sign a randomly generated bitcoin transaction using DSA. The transaction not only contains information about the bitcoin transfer, but also domain parameters and the public key of the payer. See "SingleTranscation.txt" file for details. The user will hash the first nine lines and sign the resulting hash value. The last two lines are the signature tuple.

You will be expected to provide the following delivarables:

1. A python function and its output file "DSA_params.txt" which contains q, p, and g. The format of the file must be the same as the example file "DSA_params.txt" in the attachment.
2. A python function and its output files "DSA_skey.txt" and "DSA_pkey.txt" which contain (q, p, g, α), and (q, p, g, β), respectively. The formats of the files must be the same as the example files "DSA_skey.txt" and "DSA_pkey.txt" in the attachment.
3. A python function and its output file "SingleTransaction.txt". See the example file "Single-Transcation.txt" file for details.

4. A python function that reads “SingleTransaction.txt” file and check if the signature is valid for this transaction.

We will use the attached python program “DSA_Test.py” to test the corretness of the content of the files “DSA_params.txt”, “DSA_skey.txt”, “DSA_pkey.txt’, and “SingleTransaction.txt”. For this, you need to provide code for DSA functions and generating random transaction in python files “DSA.py” and “TxGen.py”. In addition, the function names and API should match those provided in “DSA_Test.py”. For example, for DSA parameter generation your function name and its API should be

```
q, p, g = DSA.DL_Param_Generator(small_bound, large_bound),
```

where `small_bound` = 2^{256} and `large_bound`=2^{2048}.

In other words, YOU ARE REQUIRED TO TEST YOUR CODES WITH “DSA_Test.py”.

4 Phase III: Developing software for other buiding blocks and integration

In this phase of the project, you are asked to implement two Python modules that are explained in the subsequent sections. We provide you with the test file “Final_Test.py” that will be used to test your modules. Therefore, you are advised to study it carefully first.

4.1 Python module for generating random bitcoin transactions

The module name will be “TxBlockGen.py” that will include the function
`GenTxBlock(p, q, g, count)`,
which will take the public parameters of DSA (namely, p, q, g) and generates a block of transactions which contains `count` transactions. Each transaction in a block must be in the following form:

```
*** Bitcoin transaction ***
Serial number: 129986701613722743128838545549399553824
p: 12376337094338799430773844124977509457217066353 ...
q: 4534233849944221408910893818122287073031785576999072394177227217388872148783
g: 875874273754043385544666877683999314698918600107797140 ...
Payer Public Key (beta): 1064795504184295781499096314130136074379661237724834 ...
Payee Public Key (beta): 1640914200063307349695701322479808357127910275781041 ...
Amount: 905 Satoshi
Signature (r): 66067765116041166658524374136414953666714993823918402058876243 ...
Signature (s): 54222466157856912512740736834055670138544475519297459412760616 ...
```

Three files that contains eight such transactions, “TransactionBlock0.txt”, “TransactionBlock1.txt”, and “TransactionBlock2.txt” are in the attachment section of this project package.

In “Final_Test.py”, you can set the flag `TxBlocksGenOn` to generate block files, each of which contains `TxCount` transactions. Note that the number of transactions in a block must be a power of two. You are requeired to generate at least three such transaction blocks.

4.2 Python module for generating proof of work and the chain

The module name will be "PoW.py" that will include the function
`PoW(TxBlockFile, ChainFile, PoWLen, TxLen)`,
which will take the following arguments

- `TxBlockFile`: The name of the file that contains a block of transactions (e.g., "TransactionBlock0.txt" in the attachment)
- `ChainFile`: The name of the file that contains proof of works for the transactions in all past blocks. If not exits, it is created for the first block (e.g., "LongestChain.txt" in the attachment)
- `PoWLen`: the number of zero digits in the proof of work (typically set to six)
- `TxLen`: the number lines in a transaction (it is 10 in our case)

`ChainFile` includes a record for each block of transactions and each record consists of four lines:

- The proof of work for the previous transaction block
- The root hash of the Merkle Tree for the transactions in the current block (see the "bitcoin.pdf" and "bitcoin.pptx" in the attachment for Merkle tree)
- A 128-bit `nonce` value
- Proof of work for the current block

There are three flags in that you can use in this part:

- `PoWGenOn`: Lets you generate proof of work for transaction blocks in the files "TransactionBlock0.txt", "TransactionBlock1.txt", ... and the results are written/appended to the file "LongestChain.txt".
- `BlockChainTestOn`: reads the block chain file "LongestChain.txt" and validates it.
- `ValidateTxOn`: reads transaction blocks in the files "TransactionBlock0.txt", "TransactionBlock1.txt", ... and verifies the signatures for each transaction in them. Then it reads the block chain file "LongestChain.txt", picks a random transaction in a random block and verifies that it belongs to that block.

4.3 Bonus

The student group who generate the longest chain (with `PoWLen=6`) will be awarded an extra 20%. This is a competition and you do not have to participate. Include the chain length and only the last four lines of the "LongestChain.txt" file in your report. Then schedule a demo with your instructor.

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

Project Phases	**Deliverables**	**Due Date**	**Weight**
Project announcement		01/12/2017	
First Phase	Source codes, Block chain file, Readme file	08/12/2017	30%
Second Phase	DSA.py and TxGen.py DSA_params.txt, DSA_skey.txt DSA_pkey.txt , SingleTransaction.txt	15/12/2017	40%
Third Phase	TxBlockGen.py, PoW.py DSA_params.txt TranscationBlock0.txt TranscationBlock1.txt TranscationBlock2.txt ...	22/12/2017	30%
Bonus	see Bonus section	22/12/2017	20%

5.1 Policies

- You may work in groups of two (at most three)
- 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.