Advanced Cryptography

The Bitcoin crypto-currency

The objective of this lab is to discover the blockchain of Bitcoin and to implement the cryptographic primitives used in this crypto-currency.

A blockchain is a distributed ledger composed of block. In the case of a crypto-currency, each bloc is composed by an header and one or several transactions, where each transaction possess one or several inputs and one or several outputs.

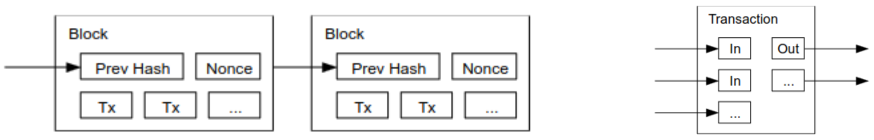

FIGURE 1 – The Blockchain of Bitcoin (Nakamoto, 2008)

The 80-bytes header of each block is composed by the block version (4 bytes), the block identifier of the previous bloc (32 bytes), the Merkle root (32 bytes), a timestamp (4 bytes), a data, called bits, describing the difficulty of mining (4 bytes) and a nonce (4 bytes). The block identifier is computed from this header. These block identifiers are used to chain the block : the blockchain is only composed of block, linked together by the differents identifiers.

The block 57043 of the blockchain posses the Bitcoin Pizza transaction used by Laszlo Hanyecz to paid two pizza with 10000 bitcoins to Jeremy Sturdivant : `https://www.coinbase.com/fr-fr/learn/crypto-glossary/what-is-bitcoin-pizza`.
This block is readable at `https://www.blockchain.com/explorer/blocks/btc/57043` and the identifier is `00000000152340ca42227603908689183edc47355204e7aca59383b0aaac1fd8`

Identify the 80 bytes of the header of the block 57043 in the web page :

1. The block version is : 0x01
2. The block identifier of the previous block is recovered in the field *Hachage* of the previous block (go to the 57042 web page) : `0x0000000013e7e85518dac94d012d73253d3fdac5c30c4143b177f3086f129580`.
3. The Merkle root is : `0x5c1d2211f598cd6498f42b269fe3ce4a6fdb40eaa638f86a0579c4e63a721b5a`.
4. The timestamp is `0x4bf81f7f`, obtained from 22 mai 2010, 20 :16 :31, by converting it with `https://www.unixtimestamp.com/`.
5. The difficulty of mining (field *bits*) is 471178276 = `0x1c159c24`
6. The nonce is 188133155 = `0xb36af23`

Verify that these data are extacly the 160 first characters (80 bytes) of the block available here (see the field *raw block*) :
`https://api.blockchair.com/bitcoin/raw/block/`
`00000000152340ca42227603908689183edc47355204e7aca59383b0aaac1fd8`
Be careful with the endianness of the data : they are represented in little endian.
Remark : the header is `010000008095126f08f ... ff84b249c151c23af360b`.

The following function will be useful in this lab :

```
def reverseBytes(data):
    data = bytearray(data)
    data.reverse()
    return data
```

Other useful functions :
`bytes.fromhex(something)` and `int.from_bytes(something, order='big')`

Download the block 57043 content in json format (*enregistrer*) and rename the file in `block_57043.json`. Write a function `recoverData` with a filename as input which returns the id of the block and the header (use `from json import load`).
For example the id can be recovered with `id_block = list(something['data'].keys())[0]`.
Write a function `checkHeader` which calls `recoverData` with `block_57043.json` and prints the block identifier.

Write a function `recoverDataFromHeader` with a 80 bytes header as input, which returns the content of this header : blockVersion, idPreviousBlock, MerkleRoot, timestamp, bits and nonce. Call this function in `checkHeader` and print the results (in little endian or not).

We have seen that the identifier of the block 57043 is :
`00000000152340ca42227603908689183edc47355204e7aca59383b0aaac1fd8`, which is the hash of the header using the `SHA256d` function (two consecutive hash with SHA256).
Write a function `checkIdBlock` with two inputs (the block identifier and the header) which returns `True` if the hash with `SHA256d` of the header is equal to the block identifier, and `False` otherwise (be careful to the little endian representation). Remark : the block identifier has the same size that the output of SHA256 (80 bytes). Call the function in `checkHeader`

The block 57043 is only composed of two transactions : the first one corresponds to the transaction fees for the successfull mining, with identifier :
`bd9075d78e65a98fb054cb33cf0ecf14e3e7f8b3150231df8680919a79ac8fe5`,
whereas the second one is the pizza transaction, with identifier :
`a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d`.
Verify these two transaction identifiers on the web site. Remark : as previously, they are represented on the website in little endian.

The next objective is the verification of the Merkle root of the block 57043. Merkle trees ara data structures ensuring the integrity of a data set. For example let d_1, d_2, d_3, d_4, d_5 be a data set and H an hash function. We compute in a first time $H_1 = H(d_1 \parallel d_2)$, $H_2 = H(d_3 \parallel d_4)$, and $H_3 = H(d_5 \parallel d_5)$, where $\parallel$ denotes concatenation. Then we compute $H_4 = H(H_1 \parallel H_2)$ and $H_5 = H(H_3 \parallel H_3)$. Finally $H(H_4 \parallel H_5)$ is the root of the Merkle tree. Write a function `CheckMerkleTree_57043` with the Merkle root and the two previous identifiers as inputs, which returns `True` if the Merkle root corresponds to the hash (SHA256d) of the two identifiers and `False` otherwise. Call this function in `chechHeader` where the two identifiers are hard coded.

The pizza transaction is described on the web page :
`https://www.blockchain.com/fr/explorer/transactions/btc/`
`a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d`,
whereas the raw transaction is available here :
`https://api.blockchair.com/bitcoin/raw/transaction/`
`a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d`
and the mining transaction is available here :
`https://api.blockchair.com/bitcoin/raw/transaction/`
`bd9075d78e65a98fb054cb33cf0ecf14e3e7f8b3150231df8680919a79ac8fe5`.
Verify that the raw data of the block 57043 is exactly : Header $\|$ 02 $\|$ mining transaction $\|$ pizza transaction, where 2 is the number of transaction.

The size of the header is 80 bytes (160 characters), the number of transaction is encoded on one byte, and, according to the web site, the size of the first transaction is 134 bytes (268 characters). Thus it is easy to recover the raw transactions from the raw block. The transaction identifier is not include in the raw transaction. This identifier is the hash (SHA256d) of the raw transaction. Write a function `checkIdTransaction` with inputs the identifier and the raw transaction, which returns `True` if the hash (SHA256d) of the raw transaction corresponds to the identifier, and `False` otherwise. Modify the function `recoverData` with a third output corresponding to transactions, and call `checkIdTransaction` in `checkHeader`.

The objective of this section is the transaction verification of the ECDSA signature. Write a new function `checkTransactions` where the signature verification is realized. It uses the Koblitz curve secp256k1, defined by the Weierstrass equation $Y^2 = X^3 + 7$ on $\mathbf{F}_p$ where p is the prime integer $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. The base point is $P = (P_x, P_y)$ where :
$P_x =$ `0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798`
$P_y =$ `0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8`
The order of P is :
$N =$ `0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141`.

The pizza transaction is complex with more than 100 inputs. The next block (57044) has also two transactions (including the mining transaction), with only one input : `https://www.blockchain.com/explorer/blocks/btc/57044`
`https://api.blockchair.com/bitcoin/raw/block/`
`0000000013ab9f8ed78b254a429d3d5ad52905362e01bf6c682940337721eb51`
`https://www.blockchain.com/explorer/transactions/btc/`
`cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79`
`https://api.blockchair.com/bitcoin/raw/transaction/`
`cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79`
The objective is the verification of this transaction (300 bytes). Recover in a first time the raw transaction from the raw block 57044 and apply the function `checkIdTransaction` to verify the transaction ID.

The signature and the public key are located in a sequence of 139 bytes called *sigscript* (you can see it on the web site with the JSON tab) :
`4830450221009908144ca6539e095 ...d6b437a8526e59667ce9c4e9dcebcabb`.

This sequence begins by three bytes `483045` which means *a sequence of 0x45 bytes* which corresponds to the signature. More precisely we have `0221`, followed by the 0x21 bytes of the integer t, then `0220` and the 0x20 bytes of s.

After the signature, *sigscript* includes the public key Q, which begins by `0141` (a sequence of 0x41 bytes), then describes the point $04 \parallel Q_x \parallel Q_y$ of the elliptic curve, where the size of Q_x and Q_y is 0x20 bytes for each one. Recover the public key Q and verify that Q is on the Koblitz curve secp256k1.

For the verification, recover the raw transaction, replace *sigscript* and the length (the byte 0x8b just before *sigscript*) by *pkscript* and the length (0x19 just before *pkscript*). Remark : *pkscript* is not located in the raw transaction, but you can see it on the blockchain explorer :
`76a91446af3fb481837fadbb421727f9959c2d32a3682988ac`.
Finally, add `01000000` at the end and hash two times with SHA-256. The hash value which should be obtained and verified with ECDSA is :
`0xc2d48f45d7fbeff644ddb72b0f60df6c275f0943444d7df8cc851b3d55782669`

There are several possibilities for *pkscript*, depending to the transaction (sometimes it countains the public key, sometimes the hash of the public key). In this transaction, verify that *pkscript* is computed from the public key $04 \parallel Q_x \parallel Q_y$ by hashing it with the SHA-256, then with the RIPEMD-160 hash function to obtain H. (`from hashlib import new` and `H = new('ripemd160')`). Finally add `76a914` before H and `88ac` at the end.

The identifier of Jeremy Sturdivant is `17SkEw2md5avVNyYgj6RiXuQKNwkXaxFyQ`. Let $Q = 04 \parallel Q_x \parallel Q_y$ be the public key. As previously, hash it with the SHA-256, then with the RIPEMD-160 hash function to obtain H. Add the null byte to H, hash two times with SHA-256 and keep the four most significant bytes. The identifier is the base 58 encoding of one null byte, H and the four previous bytes.

For debug, $H =$ `46af3fb481837fadbb421727f9959c2d32a36829`, and the four most significant bytes are `0x71c823e7`.

Modify your source code by using the `requests` library to recover directly the raw block from the website :

```
r = requests.get('https://api.blockchair.com/bitcoin/raw/block/' + id_block)
rawblock = r.json()
```