6000D-Assignment1

March 27, 2020

YANG Rongfeng, 20644943

Introduction to Blockchain Technology, BDT, HKUST

1 Q1. Hash function

Write a program or go to the reference web site to hash the phrase:

`"Hello, world!*" with a number of appended to generate 4 leading "0"s`

Reference website: https://www.xorbin.com/tools/sha256-hash-calculator

Please hash the phrase: "Hello, world!" with a number appended. For example, you can input the following phrases to see the hash result:

```
Hello, world!0
Hello, world!1
...
Hello, world!978
...
```

Can you

```
- generate 1 leading "0" ...
- generate 2 leading "0" ...
- generate 3 leading "0"s ...
- generate 4 leading "0"s ...
```

[18]:
```python
import hashlib

def find_pattern(prefix, pattern, max_num):
    for i in range(max_num):
        string = prefix + str(i)
        digest = hashlib.sha256(string.encode("utf8")).hexdigest()
        if digest.find(pattern) == 0:
            print("string = " + string)
            print("digest = " + digest)
            print("Have tried %s times" % i)
            print()
            break
```

```
prefix = "Hello, world!"
max_num = 5000
find_pattern(prefix, "0", max_num)
find_pattern(prefix, "00", max_num)
find_pattern(prefix, "000", max_num)
find_pattern(prefix, "0000", max_num)
```

```
string = Hello, world!3
digest = 098edd39515007adf089d1a8df453b5add12dde302549f4af79c5f3371cbbcc2
Have tried 3 times

string = Hello, world!67
digest = 00cde3ea31b4d24dec566dc7aff4380e1e08ce46c534517ab44d770125b3e5e7
Have tried 67 times

string = Hello, world!4250
digest = 0000c3af42fc31103f1fdc0151fa747ff87349a4714df7cc52ea464e12dcd4e9
Have tried 4250 times

string = Hello, world!4250
digest = 0000c3af42fc31103f1fdc0151fa747ff87349a4714df7cc52ea464e12dcd4e9
Have tried 4250 times
```

[22]:
```
prefixb = "Hello, world!*"
max_num = 100000

find_pattern(prefixb, "0000", max_num)
```

```
string = Hello, world!*48079
digest = 000031d7f43e78fabc1aa6535fba40c0e24d4785bff23e25543f3848173bf0ed
Have tried 48079 times
```

2 Q2. ECDSA

Show the correctness of the verification model of ECDSA signature scheme.

2.1 Signature generation algorithm

Parameter	meaning
CURVE	the elliptic curve field and equation used
G	elliptic curve base point, a point on the curve that generates a subgroup of large prime order n

Parameter	meaning
n	integer order of G, means that $n \times G = O$, where O is the identity element.
d_A	the private key (randomly selected)
Q_A	the public key (calculated by elliptic curve)
m	the message to send

The order n of the base point G **must be prime**. Indeed, we assume that every nonzero element of the ring $\mathbb{Z}/n\mathbb{Z}$ is invertible, so that $\mathbb{Z}/n\mathbb{Z}$ must be a field. It implies that n must be prime.

Alice creates a key pair, consisting of a private key integer d_A, randomly selected in the interval $[1, n-1]$; and a public key curve point $Q_A = d_A \times G$. We use $\times$ to denote elliptic curve point multiplication by a scalar.

For Alice to sign a message m, she follows these steps:

1. Calculate $e = \text{HASH}(m)$. (Here HASH is a cryptographic hash function, such as SHA-2, with the output converted to an integer.) Let z be the L_n leftmost bits of e, where L_n is the bit length of the group order n. (Note that z can be greater than n but not longer.)
2. Select a cryptographically secure random integer k from $[1, n-1]$.
3. Calculate the curve point $(x_1, y_1) = k \times G$.
4. Calculate $r = x_1 \bmod n$. If $r = 0$, go back to step 3.
5. Calculate $s = k^{-1}(z + rd_A) \bmod n$. If $s = 0$, go back to step 3.
6. The signature is the pair (r, s). (And $(r, -s \mod n)$ is also a valid signature.)

2.2 Signature verification algorithm

For Bob to authenticate Alice's signature, he must have a copy of her public-key curve point Q_A. Bob can verify Q_A is a valid curve point as follows:

1. Check that Q_A is not equal to the identity element O, and its coordinates are otherwise valid
2. Check that Q_A lies on the curve
3. Check that $n \times Q_A = O$

After that, Bob follows these steps:

1. Verify that r and s are integers in $[1, n-1]$. If not, the signature is invalid.
2. Calculate $e = \text{HASH}(m)$, where HASH is the same function used in the signature generation.
3. Let z be the L_n leftmost bits of e.
4. Calculate $u_1 = zs^{-1} \bmod n$ and $u_2 = rs^{-1} \bmod n$
5. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.
6. The signature is valid if $r \equiv x_1 \pmod n$, invalid otherwise.

Note that an efficient implementation would compute inverse $s^{-1} \bmod n$ only once. Also, using Shamir's trick, a sum of two scalar multiplications $u_1 \times G + u_2 \times Q_A$ can be calculated faster than two scalar multiplications done independently.

2.3 Correctness of the algorithm

It is not immediately obvious why verification even functions correctly. To see why, denote as C the curve point computed in step 5 of verification,

$C = u_1 \times G + u_2 \times Q_A$

From the definition of the public key as $Q_A = d_A \times G$,

$C = u_1 \times G + u_2 d_A \times G$

Because elliptic curve scalar multiplication distributes over addition,

$C = (u_1 + u_2 d_A) \times G$

Expanding the definition of u_1 and u_2 from verification step 4,

$C = (zs^{-1} + rd_A s^{-1}) \times G$

Collecting the common term s^{-1},

$C = (z + rd_A)s^{-1} \times G$

Expanding the definition of s from signature generation step 5,

$C = (z + rd_A)(z + rd_A)^{-1}(k^{-1})^{-1} \times G$

Since the inverse of an inverse is the original element, and the product of an element's inverse and the element is the identity, we are left with

$C = k \times G$

From the definition of r, this is verification step 6.

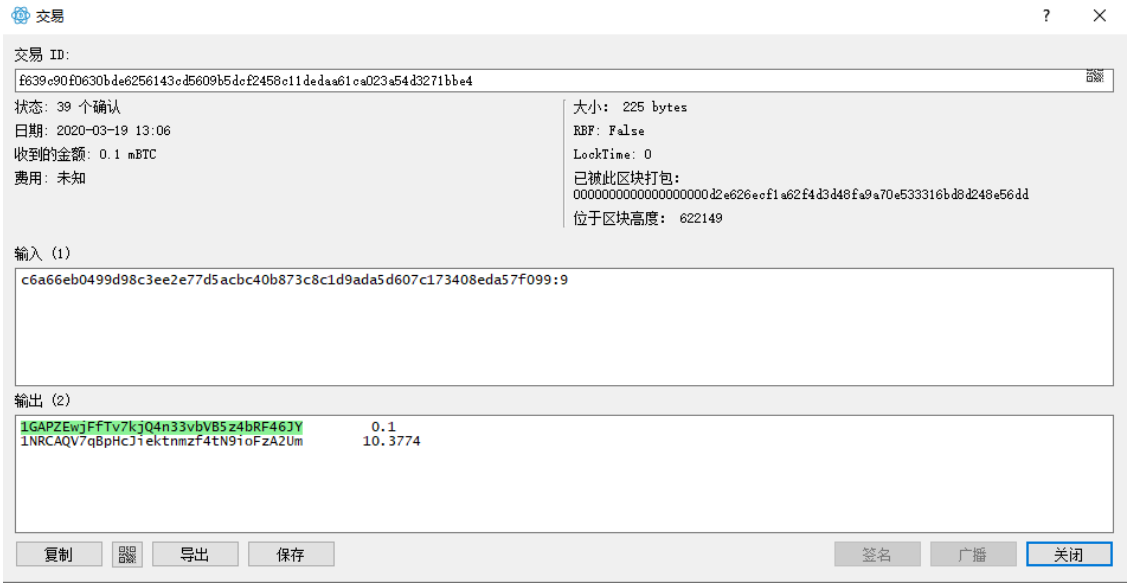
This shows only that a correctly signed message will verify correctly; many other properties are required for a secure signature algorithm.

3 Q3. Experience Blockchain

Experience Blockchain: 1. Install a Bitcoin Wallet on your PC and/or mobile phone; 2. Make some transactions (e.g. purchase something or exchange with a friend), and show the transaction record as proof; 3. List three areas of improvement for the wallet software that you use.

First, I buy some Bitcoin on Huobi Global website. Then I install a desktop wallet **Electrum** with a new account(Yes this is my first time to use Bitcoin) and I tried to transfer some Bitcoin from Huobi wallet to the Electrum wallet.

The following picture shows the transactions details: I transfer 0.1m BTC from address `1NRCAQV7qBpHcJiektnmzf4tN9ioFzA2Um` (Huobi wallet) to address `1GAPZEwjFfTv7kjQ4n33vbVB5z4bRF46JY` (Electrum wallet). Moreover, it's interesting that I only paid a few minner's fees, just 0.0226m BTC so it took me a long time to wait confirmation, about one and a half day.

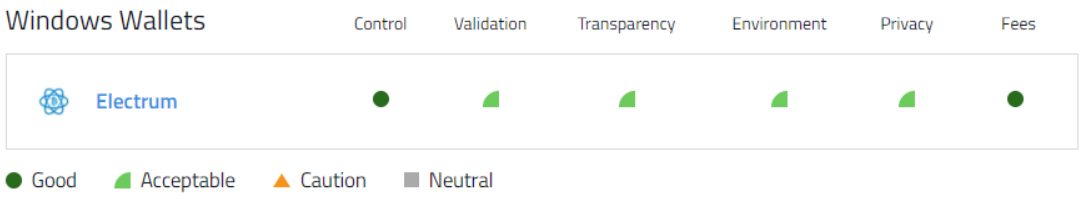


The official website of Bitcoin contains a page to guide people choose their best wallet and we compare different wallets through the website. Electrum provides you a full control of you Bitcoin property and fees that you pay during a transaction. On its official documents, it has these features:

1. *Support Two Factor Authentication.* Any users and any company with requirement of strict security can use 2FA to manage Bitcoin.
2. *Support third-party plugins.* Support Hardware wallet: *Ledger* and *Trezor* or Multi-signature.
3. *Cold Storage.* Sign transactions from a computer that is always offline. Broadcast them from a machine that does not have your keys.
4. *Light.* It's based on SPV and don't need to download the whole blockchain data.
5. *Recover.* Your funds can be recovered from a secret phrase.

However, I propose some improvement for the wallet software:

- Add the support for *Tor* as proxy to prevent others from spying your transactions by IP address
- Extend the 2FA to *multi-authentication* in some cases so it can protect Bitcoin in some insecure situation
- Some funtion of the software still imperfect. Expect it can have an embedded Bitcoin explorer. Cannot type the certain number of fees, the user only can drag the bar to choose five kinds of fees. It's not flexible.



4 Q4. Uber case study

We have discussed high traffic issues in metro cities caused by carpool services such as Uber. We want to improve the situation by using a de-centralized design. Identify three areas of problem, and propose improvement using de-centralized technical methods.

1. *Uber service increase people's willing of taking cars.* => Show information about different traffic like metro, bus, minibus, train, walking and also the combination of them. Showing the cost of time, convenience and availability. Hence, the users may not indulge in Uber way.
2. *People will gather in a certain place to get on car.* => Provide different boarding points for passengers to choose. The APP can show the waiting list on screen to encourage passengers to select areas with fewer people. Moreover, the different level of prices could be set through different aboarding points and different time to avoid people gathering at the same place or same time calling vehicles
3. *Increase the density of vehicles.* => We could make use of car pooling method i.e. passengers with same route sharing same car. This can increase the availability of cars and decrease the density of vehicles. Not only the algorithms could help us distribute passengers to various vehicles but buiding a passengers info-sharing forum in Uber is also helpful for finding partners.