Quiz4 Report

11155130 李慕庭

Problem 1

(1) Certificate Generation

Here I show my certificate file structure and explain their relationship in the following sections.

- root/
  - root.key.pem
  - root.cert.pem
- intermediate/
  - intermediate.key.pem
  - intermediate.csr.pem
  - intermediate.cert.pem
- server/
  - server.key.pem
  - server.csr.pem
  - server.cert.pem
- client/
  - client.key.pem
  - client.csr.pem
  - client.cert.pem

(2) Description of Commands

1. ./cert-go-linux-amd64-v2-3-2 create private-key -o root/root.key.pem
2. ./cert-go-linux-amd64-v2-3-2 create cert -t root -y Cfg.yml
3. ./cert-go-linux-amd64-v2-3-2 create private-key -o intermediate/intermediate.key.pem
4. ./cert-go-linux-amd64-v2-3-2 create csr -t intermediate -y Cfg.yml
5. ./cert-go-linux-amd64-v2-3-2 create cert -t intermediate -y Cfg.yml
6. ./cert-go-linux-amd64-v2-3-2 create private-key -o server/server.key.pem
7. ./cert-go-linux-amd64-v2-3-2 create csr -t server -y Cfg.yml
8. ./cert-go-linux-amd64-v2-3-2 create cert -t server -y Cfg.yml
9. ./cert-go-linux-amd64-v2-3-2 create private-key -o client/client.key.pem
10. ./cert-go-linux-amd64-v2-3-2 create csr -t client -y Cfg.yml
11. ./cert-go-linux-amd64-v2-3-2 create cert -t client -y Cfg.yml

Number 1, 2 are used to generate private-key of root and self-sign the certificate

Number 3, 4, 5 are used to generate private-key of the intermediate, generate CSR to ask certificate from the root and create certificate that signed by the root

Number 6, 7, 8 are used to generate private-key of the server, generate CSR to ask certificate from the intermediate and create certificate that signed by the intermediate.

Number 9, 10, 11 are used to generate private-key of the client, generate CSR to ask certificate from the intermediate and create certificate that signed by the intermediate.

To recreate these pem files easily, I also included a *TrustChain.sh*, which contains all 11 commands, in my zip file.

(3) Certificate Chain Description

First, the root will generate a private key and the corresponding public key, and self-sign its own certificate. In order to decrease the potential security issue of signing the certificate of servers by the root directly, usually we will create a intermediate CA. The intermediate will generate a private key and the corresponding public key, then generate a CSR to ask the root to sign the certificate. The CSR includes the public key of the intermediate and a signature by the private key of the intermediate (to prove the intermediate is an actual owner of keys). After receiving CSR and check, the root will sign the certificate for the intermediate. Similarly, the server and client will generate their private keys and the corresponding public keys respectively, then generate CSRs to ask the intermediate to sign the certificate. In the end, the intermediate will sign the certificates for the server and the client respectively. Show the sketch diagram of certificate chain in the figure 1.

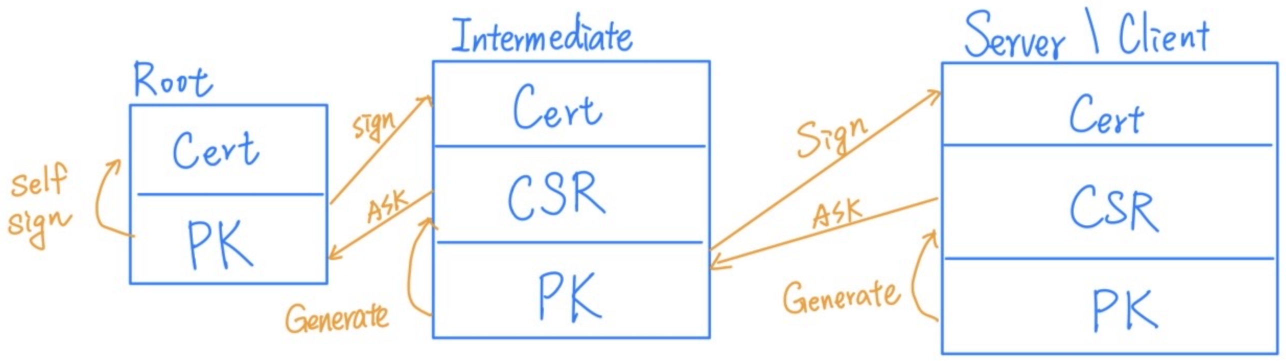
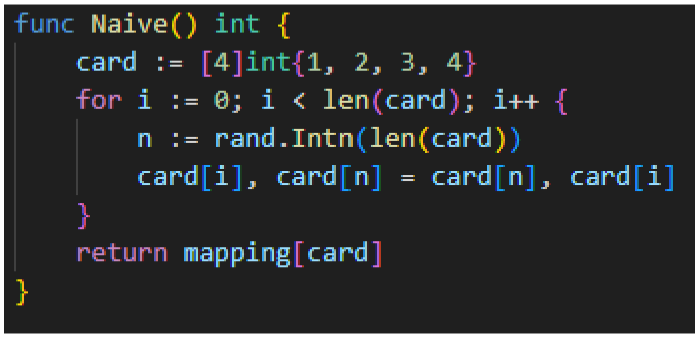


Figure 1: Diagram of Truth Chain

Problem 2

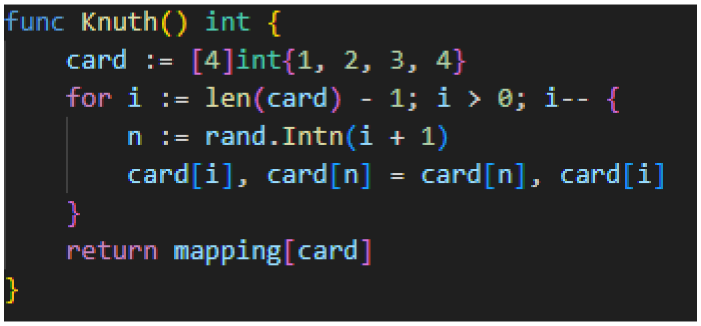
(1) Simulation Implementation

I implemented two shuffling algorithms (Figure 2). Note that, *mapping* is a map from one of permutation to a unique index between [0, 23].



```
func Naive() int {
    card := [4]int{1, 2, 3, 4}
    for i := 0; i < len(card); i++ {
        n := rand.Intn(len(card))
        card[i], card[n] = card[n], card[i]
    }
    return mapping[card]
}
```

(a) Naive

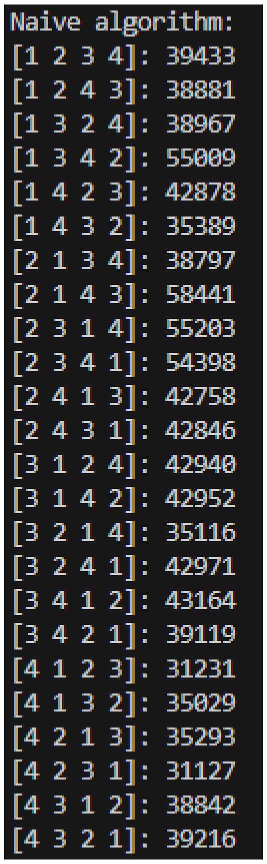


```
func Knuth() int {
    card := [4]int{1, 2, 3, 4}
    for i := len(card) - 1; i > 0; i-- {
        n := rand.Intn(i + 1)
        card[i], card[n] = card[n], card[i]
    }
    return mapping[card]
}
```

(b) Knuth

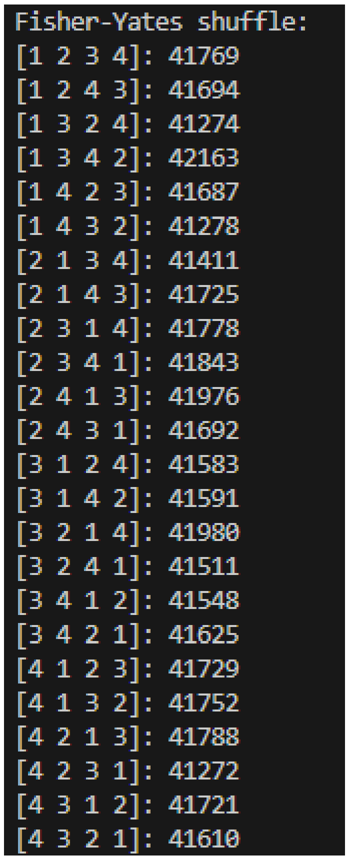
Figure 2: Two Shuffling Algorithms

After that, run each algorithm a million times and count the number of different permutations respectively. (Due to space limitation, I screenshot separately and show in Figure 3)



```
Naive algorithm:
[1 2 3 4]: 39433
[1 2 4 3]: 38881
[1 3 2 4]: 38967
[1 3 4 2]: 55009
[1 4 2 3]: 42878
[1 4 3 2]: 35389
[2 1 3 4]: 38797
[2 1 4 3]: 58441
[2 3 1 4]: 55203
[2 3 4 1]: 54398
[2 4 1 3]: 42758
[2 4 3 1]: 42846
[3 1 2 4]: 42940
[3 1 4 2]: 42952
[3 2 1 4]: 35116
[3 2 4 1]: 42971
[3 4 1 2]: 43164
[3 4 2 1]: 39119
[4 1 2 3]: 31231
[4 1 3 2]: 35029
[4 2 1 3]: 35293
[4 2 3 1]: 31127
[4 3 1 2]: 38842
[4 3 2 1]: 39216
```

(a) Naive Statistics



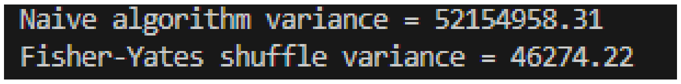
```
Fisher-Yates shuffle:
[1 2 3 4]: 41769
[1 2 4 3]: 41694
[1 3 2 4]: 41274
[1 3 4 2]: 42163
[1 4 2 3]: 41687
[1 4 3 2]: 41278
[2 1 3 4]: 41411
[2 1 4 3]: 41725
[2 3 1 4]: 41778
[2 3 4 1]: 41843
[2 4 1 3]: 41976
[2 4 3 1]: 41692
[3 1 2 4]: 41583
[3 1 4 2]: 41591
[3 2 1 4]: 41980
[3 2 4 1]: 41511
[3 4 1 2]: 41548
[3 4 2 1]: 41625
[4 1 2 3]: 41729
[4 1 3 2]: 41752
[4 2 1 3]: 41788
[4 2 3 1]: 41272
[4 3 1 2]: 41721
[4 3 2 1]: 41610
```

(b) Knuth Statistics

Figure 3: Statistics of Two Shuffling Algorithm

(2) Algorithm Comparison

Based on the statistics result, I calculate variance of two shuffling algorithms respectively. (Figure 4) I found that the variance of Naive algorithm is more than one hundred times Knuth algorithm, which means the distribution of permutations of Naive algorithm is much more unbalanced compared to the Knuth algorithm. In terms of choosing an algorithm that generates every possible permutation with average probability, the Knuth algorithm might be better.



```
Naive algorithm variance = 52154958.31
Fisher-Yates shuffle variance = 46274.22
```

Figure 4: Variance of Two Algorithms

(3) Drawback Analysis

First, we focus on the Knuth algorithm - the better one. During the first iteration, it randomly chooses from 4 numbers, and in the following iteration, randomly chooses from 3 numbers ... and so on. Multiply the possible in each iteration, it generates $4!$ possible ”swapping order”, which is identical to the number of all possible permutations.

However, the Naive algorithm, randomly chooses from 4 numbers in each iteration. In this way, the Naive algorithm will generate 4^4 possible ”swapping order”. Since $4! \nmid 4^4$, the distribution of output permutation of the Naive algorithm is unbalanced.

(4) RC4 Improvement

The insecurity issue of RC4 mainly arise from its KSA algorithm. KSA is used to shuffle the s-box (a permutation from 0 to 255) by the input key, however, it has been shown unbalance. To improve RC4, we might want to given a more random S-box shuffling algorithm. One of possible solution is used a random key stream generation (for example SHAKE-256) to generate a key stream by the input key in advance, and then used this key stream to perform shuffling algorithm. Another possible solution might be perform original KSA in several hierarchies. For example, in the first round we shuffling S-box by the input key, and in the next round we shuffling S-box by the S-box from the previous round, and so on. By applying KSA in multiple layers might decrease its random bias.

Problem 3

(1) Miller-Rabin on RSA Moduli

Miller-Rabin is a primality test by using the fact:

if $p > 2$ is a prime, then $x^2 \equiv 1$ mod n iff $x \equiv \pm 1$ mod n

Ideally, if a number is composite, it has $\frac{1}{4}$ probability to mis-classify into a prime by Miller-Rabin algorithm. Therefore, if we perform R rounds Miller-Rabin on N (R is a enough big number), we will have extremely low probability ($(\frac{1}{4})^R$) to mis-identify N as a prime

(2) RSA Security Analysis

Even Miller-Rabin is a strong **primality-test** algorithm, it cannot return **factoring** of the input number. However, to break RSA ($N = pq$), we have to know the exactly number of p and q, which cannot achieve by the Miller-Rabin algorithm. Therefore, the Miller-Rabin algorithm cannot break down RSA.