Quiz 3 Report

111550130 李慕庭

Problem 1

(1)

In terms of **structure**, SHA-256 working with 512 bits as a ”block” and 32 bits as a ”word”. At the beginning, padding input length to multiple of 512 bits and divided input into blocks, each blocks will be processed independently. Each blocks will undergo 64 rounds of processing, each round includes a series of XOR and Shift operations. And finally produce a output with length of 256 bits. On the other hand, SHA-512/256 working with 1024 bits as a ”block” and 64 bits as a ”word”. Similar to SHA-256, at the beginning of SHA-512/256, padding input length to multiple of 1024 bits and divided input into blocks such that they can be processed individually. Each blocks will through 80 rounds and obtain a output with 512 bits. And we will take first 256 bits as final outputs. In this way, we can possess the security level of SHA-512 with shorter input.

Here providing a table for briefly comparing structures of SHA-256 and SHA-512/256

	SHA-256	SHA-512/256
Length of words	32 bits	64 bits
Length of blocks	512 bits	1024 bits
Number of rounds	64	80
Length of output	256 bits	256 bits (truncated from 512 bits)

In terms of **security level**, since their length of outputs are identical, they provide equivalent security level. However, both of them use the *Merkle-Damgård* construction, exposed them to high risk of being length-extension attacks. The larger word and block size in SHA-512/256 might against this kind of attacks theoretically.

To consider **which one is better**, if we use modern computer architecture (64 bits CPU), SHA-512/256 can perform faster since its length of word is also 64 bits. Besides, the larger size of blocks can reduce the number of divided blocks and therefore decrease the number of execution rounds. Furthermore, as aforementioned, SHA-512/256 can avoid the risk of being length-extension attacks. In conclusion, SHA-512/256 might be a faster and more security choice in terms of modern computer architecture.

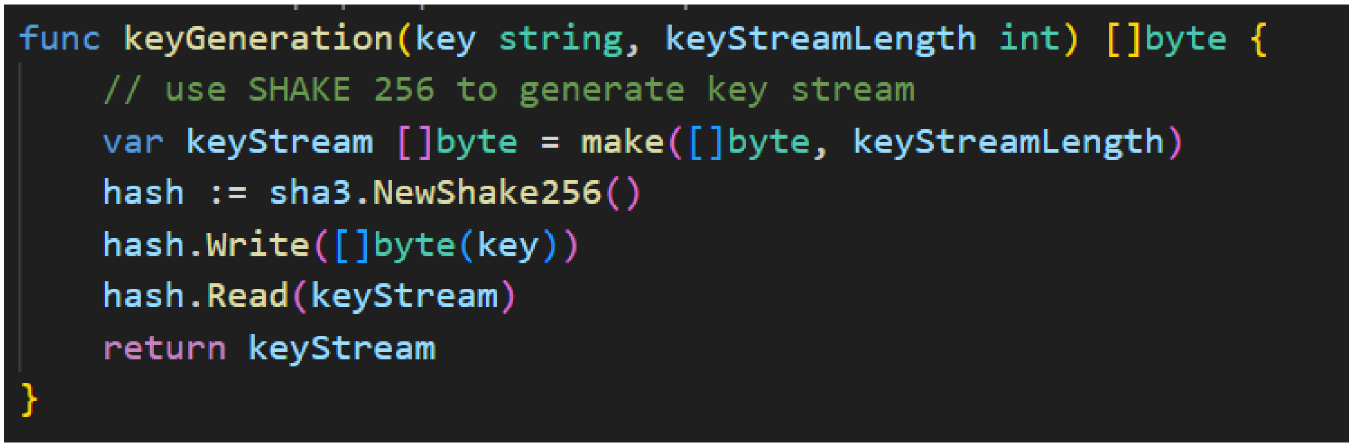
(2)

Unlike SHA-2 fixed length of output, SHA-3 based on *Keccak sponge construction*, enable to output any desired length of output. In the sponge construction, input data will process abortion and squeeze stages, which extends and compression the length of data respectively. Therefore, the length of output can be compressed into any desired length.

Due to the flexibility of SHA-3, we can apply SHA-3 to symmetric encryption by following steps. We can input a secret key and use SHA-3 to generate key stream with length equals to the length of the plain text. To encryption, we just XOR the plain text with key stream. Decryption is just the reverse process of encryption, use the same secret key to generate the same key stream and XOR with the cipher text will obtain the plain text.

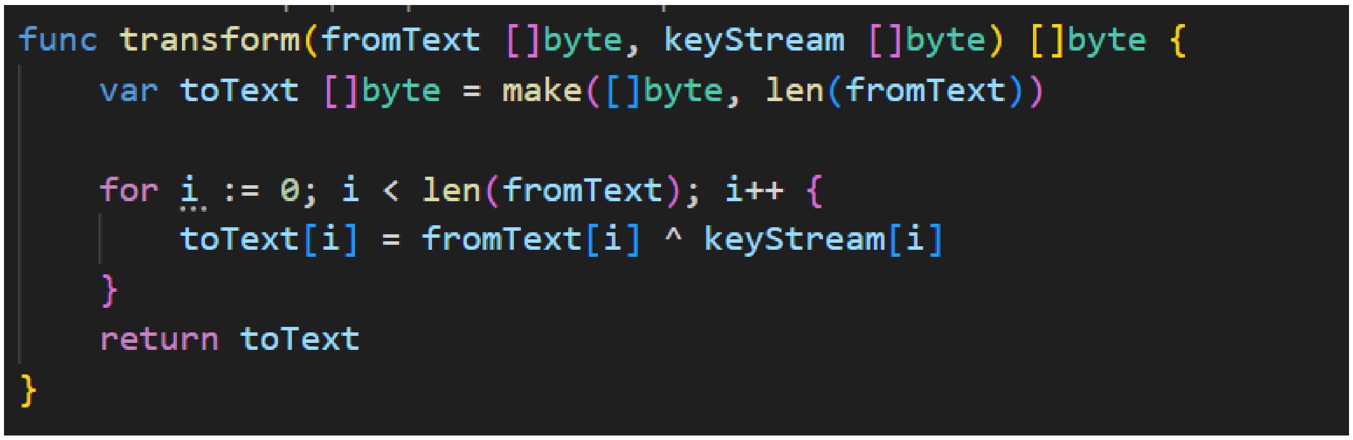
(3)

First of all, I will ask user to input a key as the seed of key stream generation and the plain text. Next I will use $SHAKE256$ as the key stream generator that input ”key” and output key stream with length equals to the length of the plain text. (figure 1) And I used a easy encryption that XOR the plain text with the generated key stream. (figure 2) Besides, to decryption, we just need to get cipher text and key. Use key to generate key stream and apply same $transform$ function on the cipher text and the key stream to obtain the plain text.



```go
func keyGeneration(key string, keyStreamLength int) []byte {
    // use SHAKE 256 to generate key stream
    var keyStream []byte = make([]byte, keyStreamLength)
    hash := sha3.NewShake256()
    hash.Write([]byte(key))
    hash.Read(keyStream)
    return keyStream
}
```

Figure 1: Generate key stream by SHAKE 256 with input key

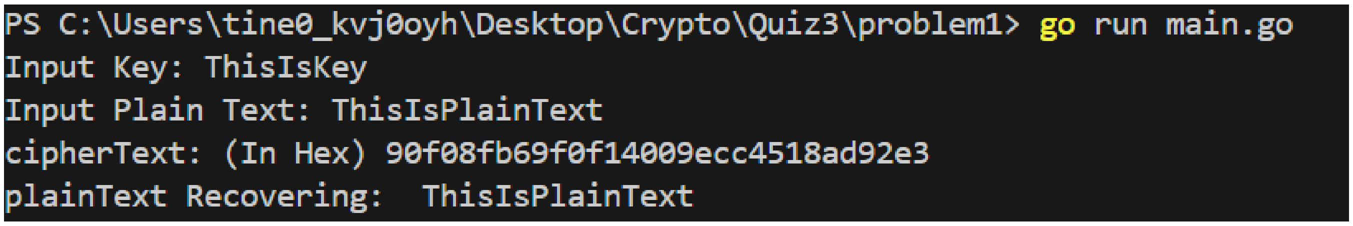


```go
func transform(fromText []byte, keyStream []byte) []byte {
    var toText []byte = make([]byte, len(fromText))

    for i := 0; i < len(fromText); i++ {
        toText[i] = fromText[i] ^ keyStream[i]
    }
    return toText
}
```

Figure 2: Transform function for encryption.decryption

Following figure 3 shows the result of demo, including input key (seed for generating key stream), input plain text, output cipher text and recovering plain text from cipher text.



```
PS C:\Users\tine0_kvj0oyh\Desktop\Crypto\Quiz3\problem1> go run main.go
Input Key: ThisIsKey
Input Plain Text: ThisIsPlainText
cipherText: (In Hex) 90f08fb69f0f14009ecc4518ad92e3
plainText Recovering:  ThisIsPlainText
```

Figure 3: The result of demo

Problem 2

(1)

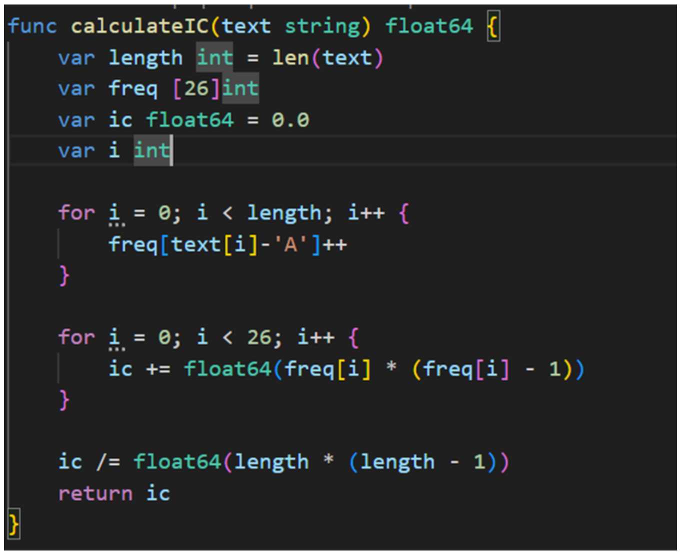
Index of coincidence calculates the probability that two randomly selected alphabets from the text are identical. N is the length of the text and f_i is the number of i^{th} alphabet in the text. The probability that randomly selected two i^{th} alphabet $= \frac{f_i(f_i-1)}{N(N-1)}$. Sum up all alphabets, we obtain

$$IC = \frac{\sum f_i(f_i-1)}{N(N-1)}$$

If the encryption method is related to mapping alphabets one-to-one, then the IC of ciphertext should be same as the IC of plaintext. We can utilize this property in many ways. For example, if we know the encryption method is one-to-one mapping alphabets (ex: Affine encryption) and the IC of ciphertext is around 0.068, we may guess the plaintext is written in English, since the IC of English is also around 0.068.

(2)

IC of cipher text ≈ 0.06810. Therefore, I guess the probable language of cipher text is English, since the IC of English is also around 0.068.



```go
func calculateIC(text string) float64 {
    var length int = len(text)
    var freq [26]int
    var ic float64 = 0.0
    var i int

    for i = 0; i < length; i++ {
        freq[text[i]-'A']++
    }

    for i = 0; i < 26; i++ {
        ic += float64(freq[i] * (freq[i] - 1))
    }

    ic /= float64(length * (length - 1))
    return ic
}
```

Figure 4: Function to calculate IC

(3)

In terms of the IC of English, Chinese and random generated text

- The IC of English ≈ 0.068.
- The IC of Chinese, unlike english is consists of 26 alphabets, there are lots of unique words in Chinese, therefore the IC of Chinese should be must lower then English.
- Assume the random generated text is written in English alphabet, i.e. it consists of 26 alphabets. N is the length of the text, then , for $1 \leq i \leq 26$, $f_i = \frac{N}{26}$

$$IC = \frac{\sum_{1 \leq i \leq 26} \frac{N}{26}(\frac{N}{26} - 1)}{N(N-1)} = \frac{N(\frac{N}{26} - 1)}{N(N-1)} \approx \frac{1}{26} \approx 0.03846$$

English only consists of 26 alphabets and the frequency of each alphabet is uneven, for example "E", "A" are more frequent then "Q", "Z", which leads to higher IC. Compare to English, Chinese consists of lots unique words and therefore has much lower IC then English. We might used IC to identify the language of plain text.

For random generated text, its IC approximately equals to $\frac{1}{C}$, where C is number of possible alphabets in the text. One of key usage is that the IC of random generated text can also be used to evaluate the performance of encryption methods. For example, if the IC of the output of certain encryption method is closed to $\frac{1}{C}$, meaning that it's more closed to random generated, which is harder for attacker to decrypt by frequency analysis or other methods.

(4)

First I try to use frequency analysis to mapping and output

```
THESEISETMOMIGNOLDORNTSWDTARBINOLTMISECENABROREMIGAN
TOUIPEATNONAUYFETHITTHESEISEOVKAOWNFGROCELADAERDAENIRC
THEOTHESMIGANTOUIPEATNODOUYFADITECTHITTHESEISEROOVKAOW
NCELADAERDAENTHELASNTUETHOCANLISUOSECALLADWFT
```

And it seems that from ciphertext to plaintext, "W $\rightarrow$ T", "H $\rightarrow$ E" and "K $\rightarrow$ H" might be correct mapping, which is identical to Caesar encryption with shift amount equals 3. Therefore, I tried to decrypt by Caesar decryption, and obtained output:

```
THEREARETWOWAYSOFCONSTRUCTINGASOFTWAREDESIGNONEWAYISTO
MAKEITSOSIMPLETHATTHEREAREOBVIOUSLYNODEFICIENCIESAND
THEOTHERWAYISTOMAKEITSOCOMPLICATEDTHATTHEREARENOOBVIOUS
DEFICIENCIESTHEFIRSTMETHODISFARMOREDIFFICULT
```

It seems meaningful. By adding proper space, I obtained the plain text:

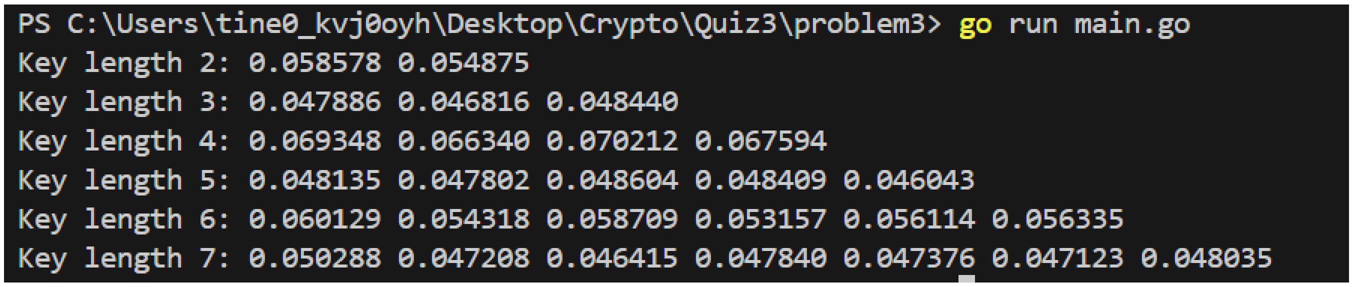
There are two ways of constructing a software design One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies The first method is far more difficult

Problem 3

(1)

I tried out all possible key length ($2 \leq keyLength \leq 7$), for each possible key length, clustered alphabets in the text with $keyLenght$ groups and calculate their IC respectively. Formally, the i^{th} alphabet in the text will be clustered into $(i \bmod keyLength) + 1^{th}$ group. For example "ABCDEFGHI" with $keyLength = 3$, I will cluster them into "ADG", "BEH" and "CFI".

By the result shows in figure 5, we might guess the key length is 4, since when key length equals 4, IC of each group is most closed to 0.068

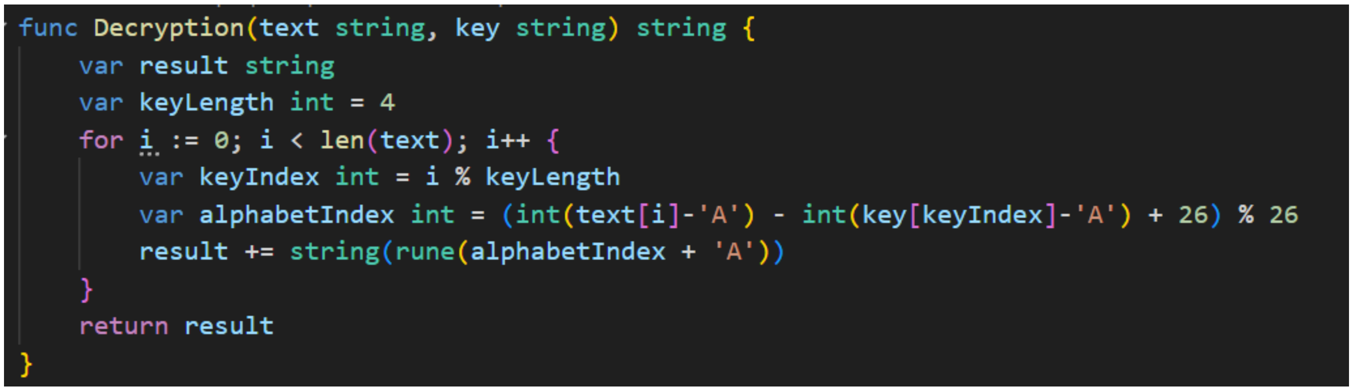


```
PS C:\Users\tine0_kvj0oyh\Desktop\Crypto\Quiz3\problem3> go run main.go
Key length 2: 0.058578 0.054875
Key length 3: 0.047886 0.046816 0.048440
Key length 4: 0.069348 0.066340 0.070212 0.067594
Key length 5: 0.048135 0.047802 0.048604 0.048409 0.046043
Key length 6: 0.060129 0.054318 0.058709 0.053157 0.056114 0.056335
Key length 7: 0.050288 0.047208 0.046415 0.047840 0.047376 0.047123 0.048035
```

Figure 5: Calculate IC of each group under different key lengths

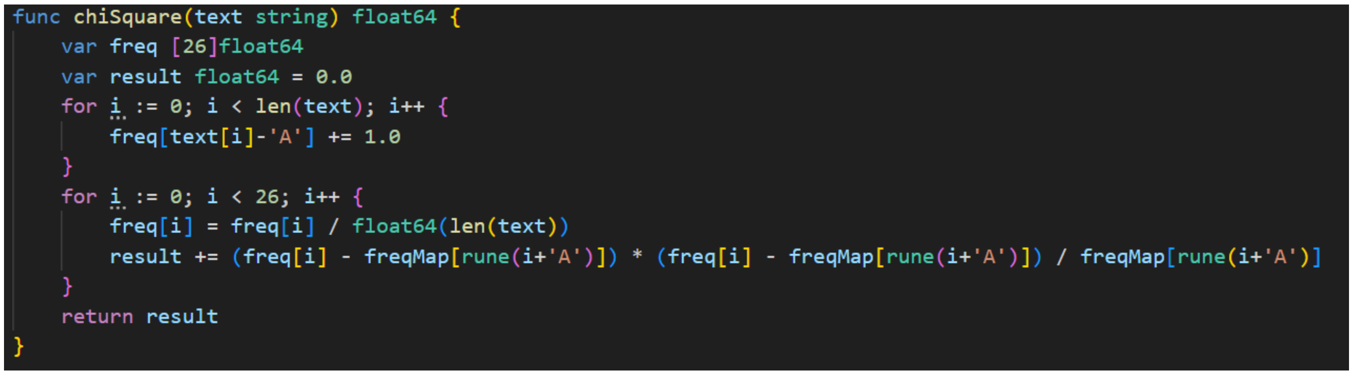
(2)

I enumerate all possible keys (total number $= 26^4$) , decryption by each key (figure 6) and calculate the chi-square value between decrypted text and given frequency map (figure 7)



```go
func Decryption(text string, key string) string {
    var result string
    var keyLength int = 4
    for i := 0; i < len(text); i++ {
        var keyIndex int = i % keyLength
        var alphabetIndex int = (int(text[i]-'A') - int(key[keyIndex]-'A') + 26) % 26
        result += string(rune(alphabetIndex + 'A'))
    }
    return result
}
```

Figure 6: Vigenère Decryption



```go
func chiSquare(text string) float64 {
    var freq [26]float64
    var result float64 = 0.0
    for i := 0; i < len(text); i++ {
        freq[text[i]-'A'] += 1.0
    }
    for i := 0; i < 26; i++ {
        freq[i] = freq[i] / float64(len(text))
        result += (freq[i] - freqMap[rune(i+'A')]) * (freq[i] - freqMap[rune(i+'A')]) / freqMap[rune(i+'A')]
    }
    return result
}
```

Figure 7: Chi-Square

(3)

By choosing decrypted text with minimum chi-square value
The decryption result : **The unanimous Declaration of the thirteen united States of America**
with encryption key is **NYCU**