Application of Number Theory in Random Number Generation to Strengthen Plaintext Encryption

Jun Ho Choi Hedyatmo 13518044[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13518044@std.stei.itb.ac.id

***Abstract*—In Cryptography, an encryption system is said to be secure if at first glance the ciphertext is very obscure such that other people shouldn't be able to identify the real message inside the ciphertext. An encryption system does two things, encryption and decryption. One algorithm or process is used to encrypt the plaintext to a ciphertext, and the other is used to decrypt the cipher text to a plaintext. In general, an encryption system usually takes plaintext and a key to produce a single corresponding ciphertext, and sometimes this is not secure enough. In this paper, we will be discussing of the possibility of an encryption system that is able to produce multiple ciphertext from a single plaintext.**

***Keywords*— cryptography, random number generator, encryption, decryption.**

I. Introduction

An encryption algorithm is a means of transforming plaintext into ciphertext under the control of a secret key. This process is called encryption or encipherment. We can write it like this

$$x = f_k(s)$$

where

- s is the plaintext,
- f is the cipher function,
- k is the key,
- x is the ciphertext.

The reverse process is called decryption or decipherment, and we write it like this

$$s = d_k(c)$$

Note that the encryption and decryption algorithm are usually public, but the main thing that makes each encryption different, using the same kind of algorithm is the private key.

There are a lot of encryption algorithm, we will present the basic and most historic one, such as the *Caesar Cipher, Substitution Cipher, Vigenère Cipher.* Other than that, we will also be discussing about the basic of number theory and random number generator that will be needed to create our own encryption algorithm. Why do *Caesar Cipher* works? How can we map each letter in the alphabet to an integer so that we can manipulate it?. It will all be clear once we learned about the mathematical part.

There are usually a lot of math involved when we talk about encryption algorithm, the whole process of encryption and decryption is basically a mathematical function which transforms It's value to another value. The encryption function do one thing, and the decryption function basically is the inverse of the encryption function. In math, It's pretty easy to see how a function is defined. And sometimes we can even find It's inverse rather quickly. But the encryption function takes a plaintext as the argument of the function which complicate some part. We need to define how a plaintext is going to be transformed. After that, the whole process just relates to a normal mathematical function which takes numbers and output numbers too. [1]

II. History

A. Shift Cipher

Encryption algorithm has been around for some time in this world. One of the earliest one ever known is called the *Caesar Cipher,* by Julius Caesar. The *Caesar Cipher* or the shift cipher involves replace each letter of the alphabet with the letter standing three places further down the alphabet. For example,

```
Plaintext:  HI IM JUNHO
Ciphertext: KL LP MXQKR
```

We shift each letter three places to the right to get the ciphertext. Here is the complete alphabet shift for the above encryption system.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
```

The shift cipher varies on how many places we're shifting the alphabet, say the amount of the shift is *k.* and we map each alphabet to a number, say from 0 to 26, from A to Z. then we can define the encryption and decryption function like this

$$x_i = (s_i + k)\ mod\ 26$$
$$s_i = (x_i - k)\ mod\ 26$$

The amount of shift is later called the *key* of the encryption. Since the numerical representation of the alphabet only span 26 numbers, the different number of possible *key* for the shift cipher is only 26. A pretty small number which can be *brute-forced* quickly so It's not very secure now, but at the time Julius Caesar invented it, it was more than enough.

B. Substitution Cipher

The main weakness with the shift cipher is that the number of keys is too small, we only have 26 possible keys. To increase the security we need more possible combination for a key to the encryption and decryption system. The substitution cipher introduces the idea of permuting the original alphabet, and mapping the alphabet to this new "permuted" alphabet, one example is as follows

```
Plaintext: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext:GOYDSIPELUAVCRJWXZNHBQFTMK
```

With the ciphertext alphabet established, we can quickly try this cipher. For example, the word `HELLO` would encrypt to the ciphertext `ESVVJ`.

The number of possible keys is equal to the number of permutation on 26 letters of the alphabet

$$26! \approx 4.03 \times 10^{26}$$

This is a very large number compared to the shift cipher key possibilities. One computer can run upto 10^8 operations in 1 second, so a *brute-force* approach would take around 4.03×10^{18} seconds.

C. Vigenère Cipher

The problem with the shift cipher and the substitution cipher was that each plaintext letter will always be encrypted to the same ciphertext letter. One way to solve our problem is to take a number of substitution alphabets and then ecrypt each letter with a different alphabet. Such a system is called a polyalphabetic substitution cipher. For example we could take

```
Plaintext: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher 1:  GOYDSIPELUAVCRJWXZNHBQFTMK
Cipher 2:  DCBAHGFEMLKJIZYXWVUTSRQPON
```

Then the plaintext letter in an odd position will be substituted with the cipher 1 alphabet while the plaintext letter in an even position will be substituted with the cipher 2 alphabet.

III. Basic Theory

In the world of mathematics, especially in number theory, we usually cares a lot about the remainder of an integer when it is divided by some specified positive integer. For instance, when we ask what time it will be (on a 24-hour clock) 50 hours from now, we care only about the remainder when 50 plus the current hour is divided bt 24. Be cause we are often interested only in remainders, we have a special notations for them. We introduced the modulo operation to reprecent the remainder of an integer. Here is the formal definition

Definition 1.1 If a and b are integers and m is a positive integer, then a is congruent to b modulo m if and only if m divides a – b. this means a and b have the same remainder when divided by m. we denote this as $a \equiv b \ (mod\ m)$

Definition 1.2 If $a \equiv b \ (mod\ m)$ and b is ranged from 0 to m – 1, we say that b form the least residue system modulo m.

Definition 1.3 If $a \equiv b \ (mod\ m)$, there exist k such that

$$a = b + mk$$

We can define arithmetic operations on these modular arithmetic too, in fact we can define addition and multiplication like this

Definition 1.5 (modular addition)

$$a\ (mod\ m) + b\ (mod\ m) \equiv (a + b)\ mod\ m$$

Definition 1.6 (modular multiplication)

$$a\ (mod\ m) \times b\ (mod\ m) \equiv (a\ .b)\ (mod\ m)$$

The addition and multiplication operation satisfy many of the same properties of ordinary addition and multiplication of integers. In particular, they satisfy these properties:

Closure if a and b belongs to the least residue system modulo m, then a + b, and a . b also belongs to the least residue system modulo m

Associativity If a, b, and c are integers, then these identities holds.

$$(a + b) + c \equiv a + (b + c)\ (mod\ m)$$
$$(a.b)c \equiv a(b.c)\ (mod\ m)$$

Commutativity If a, b are integers, then this identity holds

$$a + b \equiv b + a\ (mod\ m)$$

Distributivity If a, b, and c are integers, then these identities holds.

$$c\ .(a + b) \equiv a\ .c + b\ .\ c\ (mod\ m)$$

Identity elements The elements 0 and 1 are identity elements for addition and multiplication modulo m, respectively. They satisfy these equation

$$0 + a \equiv a + 0 \equiv a\ (mod\ m)$$
$$1\ .a \equiv a\ .1 \equiv a\ (mod\ m)$$

There are more useful things about modular arithmetic, but these basic foundation are the most important one since you can't build advanced theorem without these basic definition. One other important thing we need to address is integer representations. Integers can be expressed using any integer greater than one as a base, as we will show in this small example.

Although we commonly use decimal (base 10). Representations, other bases such as binary (base 2), octal (base 8), and hexadecimal (base 16) representations are often used, especially in computer science. In everyday life we use decimal notation to express integers. For example 324 is used to denote $3.\ 10^2 + 2.10 + 4$. However, it is often convenient to use bases other than 10. In particular, computers usually use binary notation (with 2 as the base) when carrying out arithmetic, and octal (base 8) or hexadecimal (base 16) notation when expressing characters, such as letters or digits. In fact, we can use any integer greater than 1 as the base when expressing integers. This is stated below.

Theorem 1.1 Let b be an integer greater than 1. Then if n is a positive integer, it can be expressed uniquely in the form

$$n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_1 b + a_0$$

Where k is a nonnegative integer. And a_i is nonnegative integers less than b. [2]

IV. Algorithm Construction

We are going to construct an encryption and decryption algorithm such that the ciphertext can be different each time we encrypt the plaintext, but the decryption will still yield the same plaintext. How can we do this? There are many idea regarding this, and this is called random encryption. The idea presented here will be one of the more simple one, It doesn't involve probability or anything like that. We just simply need to understand random number generation and basic number theory which we have presented in part III.

Random number generator can generate a so-called random number with a certain algorithm and seed. It's not totally random but we can definitely rely on the randomness of the generator in popular programming language such as C++, Python, Java, etc. usually a random number generator in those language will generate random number from 0 until a certain maximum value, usually the maximum value an integer can hold. But there are function to generate random number in a range too. A simple random number generator can be transformed into a range based random number generator using simple modular arithmetic. As the language that I will be using in demonstrating the algorithm is C++. I will also explain how to transform C++ random number generator to a range-based one. Here is a function written in C++ which generate a random number in a range.

```
int getRand(int a, int b){

    return (a + (rand() % (b - a + 1)));

}
```

How do the codes above works? So `rand()` will generate a random number between 0 and INT_MAX in C++, if we mod that with b – a + 1, we will get a set which is the least residue system modulo b – a + 1, which range from 0 to b – a, and if we add a to that, we get a range from a to b, as we intended.

Now that we know how to generate random number in a range, we are ready to move on to the next step. We need to generate a random number which is not so random, a random number with a pattern that we might know. How can we achieve this? There are many idea that can works. One of them is like this, we can generate a random number which will give the same remainder when we divide it with a certain value. Let's say we want to generate random number which when we divide it by 5, it will give a remainder of 3.

Let a_n be our random number, we know that

$$a_n \equiv 3 \ (mod\ 5)$$
$$a_n = 3 + 5k$$

If we let k be a random number, then a_n will also be a random number, but we know that a_n will have a special property such that when we divide it with 5, the remainder will always be 3.

Now how can we abuse this fact to make an encryption random? Actually this is the basic idea of the whole thing. We can construct a random number from a moduli m and a residue value x, and we can generate random number using the above method, and that random number is actually the ciphertext of the value x (which is the plaintext), and with the simple modulo m operator, we can reverse the random number (so-called ciphertext) back to the value x (plaintext) no matter how random it is since we construct it in a way such that the remainder will always be x. Now a random encryption is not really a function because it violates the definition of a function, but let's just say that it is. Then these are the basic definition of the random encryption and decryption algorithm.

$$e(x) = x + mk$$
$$d(v) = v \ (mod\ m)$$

Where e is the encryption function and d is the decryption function.

Now let's modify it a bit, we will construct an encryption that accept a string consisting of only alphabetic value (from a to z). and we will use this techniques to encrypt it. Let's map every value from a to z as 0 to 25. And the moduli will be 26. If we want to encrypt the word abc, we take each letter and encrypt it like this. So a is 0, let's say the random number we generate was 3, then with the above formula we can get 0 + 3 . 26 = 78 as the first number. Next, b is 1, say our random number k is 2. Then we get 1 + 2 . 26 = 53. Lastly, c is 2, say our random number is 4. We get 2 + 4 . 26 = 106. So from abc, we can get

78 53 106 we can also get 26 27 54. So the result is random. But if we want to decrypt it, we simply take each number and divide it by 26 and get It's remaider, which is 0, 1, and 2. And we convert it back to letter yielding the result abc. What's the problem with this algorithm? To differentiate each letter we need to have spaces in the ciphertext and that's not a good practice. We need something to indicate the length of each number. Let's just put their length at the front. 78 length is 2, so it becomes 278, 53 length is 2, so it becomes 253, 106 length is 3, so it becomes 3106, together without space the ciphertext would become like this

```
2782533106
```

That's a pretty good result. To decrypt it we simply get the first number, extract it until that length, and do modulo operation to it. And do it all over again.

To make it even more obscure we will introduce a key. A key should be a string too, and for each length we will add each numerical value of the key to the length modulo 26, say we're encrypting abc again with the key ghh. The numerical value of g is 6, h is 7. So we just add those to the length so the overall ciphertext becomes like this.

```
87895310106
```

Now we ran into a problem, that 10 can't be distinguished, so It's a better idea to keep it at length 1, but how can we do this? We simply encode it to a letter. So it will become like this

```
i78j53k106
```

Now we reverse each number to make it more confusing. Something like this

```
i87j35k601
```

okay, that was pretty obscure as it is, but let's transform it one more time, let each digit in even position transformed into their letter value. so the ciphertext will finally look like this.

```
i8hj3fk6a1
```

and we are done. How to decrypt it then? We simply do the reverse of what we're doing in the encryption part like this

1. Get the first letter, change it into its numerical value, subtract the numerical key value from it, and you get the length of the number.
2. After you get the length, traverse through the string until you found the string which correspond to the length.
3. Change the even position character to its numerical value and reverse the string into a number
4. Take modulo 26 at the number and convert the residue into a letter
5. Do this algorithm again for the rest of the ciphertext

And that is the decryption process. Here we will present the encryption algorithm implemented in C++.

```
string encrypt(string x, string key){
    string res = "";
    int ptr = 0;
    int lenkey = key.length();
    for(auto ch : x){
        int num = conv(ch);
        int t = num + 26 * getRand(1, 100);
        int cop = t;
        int len = 1;
        while(cop > 9){
                len++;
                cop /= 10;
        }
        len += conv(key[ptr]);
        len = (len % 26);
        res += conv2(len);
        int cnt = 1;
        while(t != 0){
                int rem = t % 10;
                if(cnt % 2 == 1){
                        res += (rem + '0');
                } else {
                        res += conv2(rem);
                }
                cnt++;
                t /= 10;
        }
        ptr = ptr + 1;
        ptr = (ptr % lenkey);
    }
    return res;
}
```

And here is the decryption algorithm also implemented in C++.

```
string decrypt(string x, string key){
    int len = x.length();
    int ptr = 0;
    string te = "";
    int ptr2 = 0;
    int lenkey = key.length();
    while(ptr < len){
        int a = conv(x[ptr]);
        a -= conv(key[ptr2]);
        a = a % 26;
        if(a < 0){
            a += 26;
        }
        a = a % 26;
```

```
            int m = 1;

            int res = 0;

            ptr++;

            for(int i = 1; i <= a; i++){

                if(x[ptr] >= 'a' && x[ptr]
            <= 'z'){

                    res += conv(x[ptr])
                * m;

                } else {

                    res += (x[ptr] -
                '0') * m;

                }

                m *= 10;

                ptr++;

            }

            res = res % 26;

            te += conv2(res);

            ptr2++;

            ptr2 = ptr2 % lenkey;

        }

        return te;

    }
```

V. Result Analysis

Let's analyze the encryption and decryption algorithm with the code above. In the above code we generate a random number between 1 and 100, and using the formula we managed to generate a random number which is always congruent to some value modulo 26. The maximum value of this is

$$MAX = 25 + 26 * 100$$
$$MAX = 2625$$

Hence, when we encrypt each character of a string, we construct the string with a maximum length of 4 (log(2625)). So the if the length of the plaintext we want to encrypt is n. then the overall time complexity of the encryption process is

$$O(4 * n) = O(n)$$

Which is linear in time. For the decryption process, the process takes the first character and convert it into integer, this is the length of the number. And then It continue to read the string until that length is satisfied. And then It repeat the steps all over again until it reached the end of the string. The whole process is also linear in time so the time complexity is also $O(n)$.

Now for all the possible combination for the ciphertext, with the same key, each letter would have 100 different encrypted value, so if the plaintext has a length of n, then the number of different ciphertext that this algorithm will produce with the same key is

$$100 \times 100 \ldots \times 100 = 100^n = 10^{2n}$$

For a single plaintext with length 3, the encryption algorithm produce a million different ciphertext, if we also consider the different key possible, with regard to the plaintext length, we have 26^n different combination for the key. So the total combination possible is

$$26^n \times 100^n = 2600^n$$

Which is a really big number for big n since it expand exponentially.

The program above has been tested and we generate 20 different ciphertext using the algorithm for the plaintext `helloworld` using the key `junhochoi`. Here are the results

```
n1j1cy6g2cr9d0ck7c4s8f4cf8f9l6h2cs5d4cm1b3bm5h5
n5e6bx2d7r7h3cl1h5bs8h6bf4c7l0i0br9b7m9a9bm5g9
m9b3y8j7br9h7bk1j7s2h1cf8c8l2f6br1e6m9h7bm9d9
m7i7w2iq9a6l5e5bs4d8bg0j7bl8f4cs9j4bl3c3n1c1b
m7i7y8c9bq3i5k1d5s4c2cg0b0bl8i5cr7g6l7c4n9e8b
m1h3x6a7r3a4cl7b1cs6j4bg6e9bl2f6bs3g8bm3d5cn5d1c
m3g1x0i6r1c2ck9d7r6h9f8h1l6d2br7j7l5h3n1b5b
m7f6y0a5cr9g1cl7a2br4a4g0j7bk4c9s7a7bm5f1bn9c3b
n5j2cx8d2r3b0cl5g0cs8c3cg2d2ck8b1s9g3bm1e4bn3a6c
m5g8y0j5br3b0cl5a8bs4f0bf0g3k6b7s9h2cm1i4cn3e0b
n3a5cy0f8bq1c9l5g0cs2i7be4hl2g5cq5jl9h4n1c4c
n5i3by6i4bp3gl7b1cs6h2cg4b1bl0d7bs7f3cm1i1bm3c5
n3h0bx4i7q3e8k9h4s2g5cg4j8bl6e1cs3j9bl1h2m1d7
n5h7by6i4br9f5cl5b4bs6h2cg8d7bk6j1s1a2cl1d5m3f6
m1g7y6h8bq9g8k1j7r8b1f2e5l0g8bs1b8bl5e2m9b4
n3h0by4a3br3d5ck5d6r8a5g8b2bl6d2bs9g3bl3i5n9f4b
n7i0cy8b3cr1a7bk5g7s6a1bg8e3bl8f4cq3el1c9n3j6b
m7f6x4j3r9j2cl3j4bs8f1bg4b4ck2e7s3d7bm1h5bm5g9
m5g8y6f3bq9j9k5g7s0j9bf0d2l4g9br9i5m5e5bn9j4c
m7i7x2j9q7e9l5f4cr8a5f6h7l8f1br9c3l5g7m3i7
```

and then decrypting all 20 of them will yield the same result, helloworld.

In retrospect, this encryption system can provide some level of security than just a normal cipher. But this encryption system can still be analyzed and people can crack it with statistics, we can actually see that with the same key, the frequent letter will be the same, and we can analyze it even further by dividing it into chunks.

This encryption system also has a few more weaknesses. Such that it can only accept lowercase alphabet (a – z). it can be expanded to include uppercase alphabet (A – Z) and numbers (0 – 9). But we need to map the alphabet to a bigger residue system, not just 26. But we also run into the problem where non alphanumeric character might be included, like {, (, #, etc. this needs more analysis as we need to decide what to do with them. Changing them into their ASCII code might work, but we need to consider the decryption process too if the ciphertext ends up being indistinguishable.

VI. Conclusion

In this paper we discuss of the possibility of constructing an encryption system where the encryption process will yield more than one ciphertext, and where the decryption process will always yield the same plaintext (using the correct key, of course). Turns out It's possible just using basic number theory, basic definition of modular arithmetic, etc. the use of randomness in an encryption algorithm like this is probabilistic encryption. But the way we implemented things were different than an actually probabilistic encryption.

The encryption and decryption algorithm described in this paper is just a basic one that is dependent on the fact that we could create random number which is not so random, such that each random number remainder when divided by a certain number will always be the same. Other method can be used to construct a function which yield different result each time, but the inverse is always the same. We could use fermat's little theorem, modular inverse, etc. but this algorithm with the use of algebraic to modular form using a bit of random number generator is probably simpler than using all of the aboves theorem.

VII. Acknowledgment

The completion of this paper could not have been possible without the help from the Almighty God. The author thanks Ibu Harlili as the lecturer of IF2120 Matematika Diskrit Class 2. The author also thank his family, friends, and many more people for always supporting him. The author also thank Bandung Institute of Technology for giving him the opportunity to learn and grow much more.

References

[1] Nigel Smart, Cryptography: An Introduction (3rd Edition), pp.37-47
[2] Kenneth. H. Rosen, Discrete Mathematics and Its Application (7th edition), pp.240-246

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 4 Desember 2019

[illegible]

Jun Ho Choi Hedyatmo 13518044