Misinterpretation Cipher

IRVAN JAHJA / 13509099
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13509099@std.stei.itb.ac.id

March 15, 2012

Abstract

This paper describes an alternative to the usual encryption algorithms that grants no information to adversaries on some circumstances and conditions. We present two possible implementations of the cipher as well as examples on the usage of this cipher.

1 Introduction

1.1 Background

The flow of current modern researches on cryptography mainly focus on the encryption of one plaintext. This is indeed the most natural question posed for cryptography, to safely send a message to another party in such way that only the two involved parties are able to understand the message.

Unfortunately, such techniques are prone to cryptanalysis techniques. Many cryptanalysis techniques relies on finding a key for which the plaintext 'make sense', for instance by assuming that the plaintext is an english text. The success of such algorithm is based on the fact that each ciphertext is constructed from exactly one plaintext and so, if they found a key for which the plaintext is an english sentence, the probability that that is the actual plaintext is astronomically high.

1.2 Alternative

It is then makes sense to ask whether we can encrypt more than one plaintexts inside a single ciphertext. For some plaintexts, cryptanalysis will never be succesful without knowing the actual key, since now there are a multitude of possible plaintexts and the question of "which one of them is the real plaintext" cannot be answered with absolute certainty by the adversary.

For the rest of this paper, we will call this technique the misinterpretation cipher.

1.3 Usage

This kind of cipher is most usable to encrypt a text that conveys a choice. For example, we can use this kind of cipher to encrypt the result of a yes-no question. We encrypt in each ciphertext both "YES" and "NO". Adversaries that decrypts the message will obtain both "YES" and "NO", from which they can gain no conclusion on the real message.

There are of course some situations on which this cipher is not ideal. Password encryption is an example: a password encrypted using this cipher means that the adversary upon cryptanalysis will obtain a small finite set of possible passwords, from which he/she can try by brute force the passwords to see which of them is the correct password.

However, the virtue of this cipher is that it is extremely simple to implement. Indeed, we will show that the simplest possible implementation is simpler than the infamously weak Caesar Cipher. However, we have argued that without knowing the key, it is not possible based on Cryptanalysis alone to know which of the possible plaintexts is the real plaintext.

Another virtue is that the key required is very small compared to other respectable ciphers such as One Time Pad. Indeed, in one plaintext ciphers, the key must be inherently based on an enormous domain for otherwise an adversary can easily guess the key by brute force. This is not possible in our case, since there will be more than one keys that is consistent with the given cipher-

text.

1.4 Roadmap

This paper is organized as follows. Section 2 will discuss a very simple implementation of this algorithm, and then presents an implementation of the algorithm in Python as well as some examples on encryptions for ciphertext and plaintext. Section 3 will discuss another idea based on Chinese Remainder Theorem. We will present working implementation of the algorithm as well as example usage of the algorithm.

2 A Very Simple Misinterpretation Cipher

2.1 Idea

When you are asked to implement a misinterpretation cipher, the most immediate solution is to simply concat the two strings. Indeed, this is a possible implementation but suffers from the concept of "fairness", that is, if the key is streamed, one plaintext will be available far before both plaintexts are available. The natural amendment is to simply interleave the two plaintexts.

2.2 Encryption Algorithm

- Set up a constant N, denoting the number of plaintexts to encrypt in a single ciphertext.
- Prepare N identical-length plaintext (by appending characters if necessary)
- Randomly shuffle the plaintexts, and let the key be the index of the actual plaintext in this ordering
- Interleave the plaintexts character by character in order of the shuffle
- Send this as ciphertext.

2.3 Decryption Algorithm

- Receive the ciphertext
- Construct the plaintext by picking only the characters in the the positions X such that X modulo N = key.
- Return the plaintext

2.4 Implementation

Figure 1 presents an implementation of the encryption algorithm and the decryption counterpart is presented in Figure 2.

Note that most of the code is spent on appending '#' to make the length of the plaintexts equal: under the assumption the plaintexts are of equal length, the code becomes very short.

2.5 Example Usage

We exemplify this by trying to send the message conveying the time of attack. The real plaintext is "DAWN", and we will use key = 2 and N = 3, the other two misleading messages being "NIGHT" and "EVENING". We do this by issuing the following command:

```
encrypt([ 'DAWN' , 'NIGHT' , 'EVENING' ] ,2)
```

The result is 'NEDIVAGEWHNNTI##N##G#'. Encryption is performed by calling the following function:

```
decrypt( 'NEDIVAGEWHNNTI##N##G# ' ,3 ,2)
```

which returns 'DAWN'. Note that indeed, the letter 'D' is at position 2, equal to our key.

2.6 Complexity Analysis

The time required for both encryption and decryption is O(N), where N is the total length of all plaintexts.

3 Chinese Remainder Theorem based Algorithm

3.1 Introduction

Chinese Remainder Theorem is as follows. Suppose $n_1, n_2, \ldots, n_k$ are positive integers which are pairwise coprime. Then, for any given sequence of non-negative integers $a_1, a_2, \ldots, a_k$, such that $a_i < n_i$ there exists a unique non-negative integer $x < \prod_{i=1}^{k} n_i$ solving the following system of simultaneous congruences.

$x \equiv a_1 \pmod{n_1}$
$x \equiv a_2 \pmod{n_2}$
$\cdots$
$x \equiv a_n \pmod{n_k}$

That is, $\forall z : 1 \leq z \leq k \rightarrow x \equiv a_z \pmod{n_z}$.

Readers interested in the proof are referred to [2].

```
# Encrypts a set of plaintexts with the given key. plaintexts[0] is the
# real plaintext
def encrypt(plaintexts, key):
    # First, make all plaintexts have the same length
    max_len = 0
    for plaintext in plaintexts:
        max_len = max(max_len, len(plaintext))

    pt = []
    for i in range(len(plaintexts)):
        plaintext = plaintexts[i]
        ptcopy = plaintext
        while len(ptcopy) < max_len:
            ptcopy = ptcopy + '#'

        if i == 0:
            special = ptcopy
        else:
            pt.append(ptcopy)

    # randomly shuffles the rest of the plaintexts
    random.shuffle(pt)
    pt.insert(key, special)

    # interleave it
    ciphertext = ''
    for i in range(max_len):
        for p in pt:
            ciphertext = ciphertext + p[i]

    return ciphertext
```

Figure 1: Simple Interleave Encryption implementation in Python

```
# Decrypts a given ciphertext
def decrypt(ciphertext, N, key):
    plaintext = ''
    for i in range(len(ciphertext)):
        if i % N == key and ciphertext[i] != '#':
            plaintext = plaintext + ciphertext[i]
    return plaintext
```

Figure 2: Simple Interleave Decryption implementation in Python

In this section, we will represent plaintexts as a single integer. The justification for this is that we can represent string uniquely as an integer by transformations. For example, one possible transformation of lower case characters is by representing the string as a base 26 number.

3.2 Idea

First, the sender and the receiver will agree on the number of plaintexts encrypted in a single transmission. We will call this number k. Then, they must agree on k positive pairwise coprime numbers $n_1, n_2, ..., n_k$. Each of these numbers must be greater than then largest possible plaintext. Finally, they pick one of these numbers as the key. The size of the key can be made small by sorting the coprime numbers and storing the index of the key.

3.2.1 Encryption

We assume that the plaintexts to encrypt are $p_1, p_2, ..., p_k$ and that if the key is y, the p_y is the real plaintext.

The algorithm find $0 \leq x < \prod_{i=0}^{k} n_i$ such that the following simultaneous equations hold: $x \equiv p_1 \pmod{n_1}$

$x \equiv p_2 \pmod{n_2}$

...

$x \equiv p_k \pmod{n_k}$

That is, $\forall z : 1 \leq z \leq k \rightarrow x \equiv p_z \pmod{n_z}$.

The Chinese Remainder Theorem guarantees its uniqueness and there are known polynomial time algorithms to compute such number.

The algorithm then proceeds by sending x as the ciphertext.

3.2.2 Decryption

Decryption is extremely simple. Using the agreed key y, it proceeds to obtain the plaintext from the ciphertext x by the following operation:

$p = x \bmod n_y$

3.3 Implementation

Figure 3 and 4 implements the two functions needed by this algorithm, namely Extended Euclid and Chinese Remainder Theorem. Figure 5 embodies the encryptor. The constructor takes as its parameter the k numbers n_1 through n_k that has been agreed upon both parties.

Note that encryption does not require key as it is assumed that the real plaintext is located as the key-th plaintext provided.

3.4 Example Usage

For the sake of simplicity, we will take a contrived set of numbers to encrypt which are not necessarily the result of the encryption of a set of strings.

Our example will use N = 3 with the values of n_1, n_2, n_3 as three prime numbers 983, 991, and 997. Note that the algorithm does not require the numbers to be primes, it only require that the numbers are coprime to each other.

The encryptor will be build as follows:

```
x = CRTEncrypt([983,991,997])
```

To encrypt a set of messages, we will call the .encrypt() method of CRTEncrypt. For instance, to encrypt messages 400, 500, and 600, we will call the following function:

```
cipher = x.encrypt([400,500,600])
```

In our example, the resulting ciphertext is: 335583821.

To decrypt the message, we will call the .decrypt() method. Assuming that 500 is the real plaintext, we call the function as:

```
plaintext = x.encrypt(335583821,1)
```

Note that in our implementation, the key is 0-based. The result of the method invocation is 500, which is consistent with what we are looking for.

3.5 Complexity Analysis

We will describe the complexity in term of the number of bits required to represent all the k numbers feed into the encryptor as n_1 through n_k. Assume it's N.

Decryption consists of a single modular operation. Using modern techniques such as FFT, a complexity of $O(NlogN)$ can be achieved.

Encryption is more complex. Multiplicating all k numbers n_1 through n_k takes time proportional to $O(NlogN)$, using FFT for instance. Adding the summand of x takes $O(NlogN)$. Finally, the Extended Euclidian invoked k times contributes to $O(k * N)$ time. Hence, the total complexity is $O(k * N + NlogN)$.

```
# Extended Euclid implementation
def ee(a, b):
    orig_a = a
    orig_b = b
    a = abs(a)
    b = abs(b)
    swapped = 0
    if a < b:
        (a, b) = (b, a)
        swapped = 1

    if b == 0:
        if swapped:
            return (0, 1)
        else:
            return (1, 0)

    s0 = 1; s1 = 0; t0 = 0; t1 = 1
    r0 = a; r1 = b

    while (True):
        q = r0 // r1
        r2 = r0 % r1
        if r2 == 0:
            if orig_a < 0:
                if swapped: t1 *= -1
                else: s1 *= -1
            if orig_b < 0:
                if swapped: s1 *= -1
                else: t1 *= -1
            if swapped: return (t1, s1)
            else: return (s1, t1)

        s2 = s0 - q * s1
        t2 = t0 - q * t1
        s0 = s1
        s1 = s2
        t0 = t1
        t1 = t2
        r0 = r1
        r1 = r2
```

Figure 3: Extended Euclid implementation

```
# Chinese Remainder Theorem implementation
def crt(a, n):
    prod = 1
    for i in n:
        prod *= i

    x = 0
    for i in range(len(a)):
        val = ee(n[i], prod // n[i])
        x += val[1] * a[i] * (prod // n[i])

    while x < prod:
        x += prod
    x %= prod
    return x
```

Figure 4: Chinese Remainder Theorem Implementation

```
class CRTEncrypt(object):
    def __init__(self, n):
        self.n = n

    # Assumes p are integers.
    def encrypt(self, p):
        return crt(p, self.n)

    def decrypt(self, cipher, key):
        return cipher % self.n[key]
```

Figure 5: Chinese Remainder Theorem misinterpretation cipher

3.6 Possible Improvement

We let any set of numbers n to be used. However, it might be much more beneficial to use large prime numbers as n. This is so since factoring integer can yet be solved efficiently [1]

4 Conclusion

Misinterpretation cipher has as its virtue simplicity in implementation while at the same time maintaining the property that in its use case, it is virtually unbreakable.

The major drawback of this cipher is the size of ciphertext is virtually multiplied by the number of piggybacked messages. However, it is easy to see that in the assumption that all possible combinations of plaintexts is possible to be inputted by the user, we can do no better than this.

References

[1] Richard P. Brent. Recent progress and prospects for integer factorisation algorithms, 2000.

[2] Kenneth H. Rosen. *Discrete Mathematics and Its Applications, 5th Edition*. 2003.