Coursework 2 - One Time Pad Attack

Rafael Rost

***Abstract*—In order to complete a coursework of master degree cryptography classes, in this report I have presented how to decrypt a series of messages that was encrypted using One Time Pad cipher, by taking the advantage of key reutilization. As part of the cryptanalysis work, a tool was developed using Ruby language. The tool did not decrypted all the messages, but was very helpful and saved me a lot of hard working. Source-code is available on my GitHub account.**

Keywords*—*One Time Pad, Ruby, Cryptography.

I. Introduction

In theory, One Time Pad is a Stream Cipher that is consider perfectly secure. Basically, this cipher encrypts messages using a unique key that has the same size of the message. In other words, each message byte is encrypted with relative key byte.

However, this cipher is only secure if is properly used. If for some reason, the same key is used to encrypt more than one message, there is a big chance to perform a cryptanalysis and reveal messages content.

II. One Time Pad

A. Understanding the Algorithm

The encrypt/decrypt algorithm behind OTP is very simple. The cipher-text is the result of a XOR operation between message and key, like we can see in this example:

- Message : 0 1 0 1 1 0
- Key : 1 1 1 0 0 1
- Cipher-text : 1 0 1 1 1 1

One of the advantages of this cipher, is that it is really fast to decrypt/encrypt messages. But, the big problem is that the key must have the same size of the message, and sometimes this turns OTP an impractical cipher.

B. XOR Operation

Since the main OTP cipher operation is the XOR, lets try to understand a little bit more about this operation. XOR operation basically returns true whenever the two values differs and false when they are equals. We can see this in the XOR truth table:

- 0 ˆ 0 = 0
- 0 ˆ 1 = 1
- 1 ˆ 0 = 1
- 1 ˆ 1 = 0

One important thing that we can notice on this truth table, is that is possible to recover the values that were xored together. For instance, xoring 1 with 1 give us 0 as result. Xoring the result (0) with 1, give us 1 again. This show us that xoring the cipher-text with the key, will give us the plaintext, and it possible to check this by inverting the truth table:

- 0 = 0 ˆ 0
- 0 = 1 ˆ 1
- 1 = 0 ˆ 1
- 1 = 1 ˆ 0

III. Cryptanalysis

The cryptanalysis process was done in two parts. The first one, was performed in an automated way, using a tool that was coded using Ruby, and it decrypted almost entire cipher-text. The second part was done by hand, and it was just fill or correct some words into the text, by using obvious guesses. But, in order to confirm the guesses, I ran the tool again hard coding the key, and this showed the entire plaintext and key.

As part of this coursework, we also received a hint about the cryptanalysis process: Consider what happens when something is xored with space (hex 32). So, lets check that, by playing with some Ruby code.

```
# XOR empty space with a..z
xored = ''
'a'.upto('z').to_a.each do |c|
  xored += (c.ord ^ ' '.ord).chr
end
puts xored
```

- Result: ABCDEFGHIJKLMNOPQRSTUVWXYZ

```
# XOR empty space with A..Z
xored = ''
'A'.upto('Z').to_a.each do |c|
  xored += (c.ord ^ ' '.ord).chr
end
puts xored
```

- Result: abcdefghijklmnopqrstuvwxyz

So, basically when you XOR an letter with a space, the letter will swap case, which means, an A becames an a and vice-versa. Also, lets check what happens when we XOR a space with another space.

```
# XOR two empty spaces
puts (' '.ord ^ ' '.ord).chr
```

Result: ' '

Xoring two spaces, gave us another space. With this hints on mind and also knowing a little bit about the XOR operation, we have now food enough to start performing the cryptanalysis process. The next session of this paper will describe how the cryptanalysis was implemented on the ruby decrypt tool.

A. Xoring cipher texts together

At this time, all that we know is that a list of messages was encrypted using the same OTP key. So, lets now imagine this scenario:

- Message1: hello
- Message2: devil
- Key : abcde

In this case, the first cipher-text will be:

- (h ˆ a) (e ˆ b) (l ˆ c) (l ˆ d) (o ˆ e)

And the second one will be:

- (d ˆ a) (e ˆ b) (v ˆ c) (i ˆ d) (l ˆ e)

Lets see what happens when we XOR this two cipher-texts together.

- (h ˆ a) ˆ (d ˆ a) = (h ˆ d) ˆ (a ˆ a)

Now, lets play with ruby again, and see what happens when we XOR a value with itself.

```
# XOR value with itself
puts ('A'.ord ^ 'A'.ord).chr
```

- Result:

We got NULL! So, we can say that our equation could looks like this:

- (h ˆ d) ˆ NULL

Finally, we can check the result of a XOR between a value and NULL:

```
# XOR value with NULL
puts ('B'.ord ^ ('A'.ord ^ 'A'.ord)).chr
```

- Result: B

Like we can see, we got the value itself as a result, which means that the final result of our equation will be:

- (h ˆ d)

Doing the same with all the message letters, we will have this:

- (h ˆ d) (e ˆ e) (l ˆ v) (l ˆ i) (o ˆ l)

Nice! We ended up with a new cipher text, the difference of this one is that we removed the key from it!

We also learned that a XOR between space and letter will swap case the letter. So, we can look at the new cipher text and assume that every character between [a-zA-Z] is the product of a xor between a space and another character. We can also assume that all spaces are the product of two another spaces.

At this point, by xoring two cipher texts encrypted with the same key, we are able to identify that one of them have a space on a specific position. Lets now discover which one has that space.

B. Discovering empty spaces

Supposing that we have this messages A and B, which were encrypted using OTP:

- A: hello my friend
- B: give me the key

Xoring the cipher texts of this two messages gave us something like this:

- * * * *_ _ * _ _ * * _ * * *

Since we are only interested in the spaces, I represented all another cipher texts characters with *. We know that messages A or B have a space on the positions represented by _. Now, lets introduce a new message to our scenario, and XOR it with message A.

- A: hello my friend
- C: hey friend

The result of XOR between the cipher texts of this two messages will be:

- * * * _ * _ * * _ * * * * *

Since we are comparing message A with another two messages, we can consider that every space found on same position in two comparisons, must belong to message A. Look at the result:

- * * * * _ _ * _ _ * * _ * * (A with B)
- * * * _ * _ * * _ * * * * * (A with C)
- * * * * * _ * * _ * * * * * (common spaces between comparisons)

Now we discovered exactly which spaces belongs to message A. Off course, this method is not so precise, because we can have spaces at the same positions at B and C messages, and this can drive us to some false positives. However, the more messages we have, more comparisons we can perform and more precise our result will be. And thats the case of this coursework. On the ruby tool, I rotated all the messages and did this comparisons between each message with all another messages. The result of this, was that I discovered all the spaces to all the messages.

C. Finding the key

Knowing all the spaces of all the messages gave us the ability to access the key. We simple XOR the message spaces positions with a space and this will reveal the key. At this point, the automated cryptanalysis was done, and we ended up with this plain text:

m0: CAESA_ A_D _IGENERE _A_E CIPHER___HA_ _CA_ B__ E_SI__ BR_____

m1:THIS IS _HE _ECON_ STRING_T_AT WAS ___RY_TE_ _SI__ O_E __ME _____

m2:ONE TIME _AD W_LL P_OVIDE A _I_HERTEXT O___ A_TA_K _EC__ITY_

m3:IN THE C_IPTO_RAPH_ CLASSES_Y_U WILL LE___ H_W _O _ON__SE _ND__O DIF_____

m4:TO UNDERS_AND _ISCR_TE PROBAB_L_TY IS IMP___AN_ T_ U_DE__TAN_ C__PTOGR_____

m5:ENGLISH L_TTER_ FRE_UENCY ALL_W_ YOU TO B___K _IG_NE_E __D C_ES__ CIPH____

m6:CRYPTOGRA_HY I_ PRE_ENT IN SE_E_AL DIFFER___ T_PE_ O_ A__LIC_TI__S NOW______

m7:IF YOU FO_ND T_E ON_ TIME PAD_ _HEN YOU H___ F_NI_HE_ T__S _OU__EWORK_

m8:Every clo_d ha_ a s_lver lini_g_

This is awesome! As you can see, now its a piece of cake to guess and fill the missing words inside the messages.

D. Guessing

Now, that we have a partial key, and also a good idea of the plaintext, lets perform some guesses in order to discover the entire key. Look at message seven partial plain text: IF YOU FO_ND T_E ON_ TIME PAD_ _HEN YOU H___ F_NI_HE_ T__S _OU__EWORK_.It looks like we can have a good guess for the entire message: IF YOU FOUND THE ONE TIME PAD THEN YOU HAVE FINISHED THIS COURSEWORK.

Good guess right? First empty letter that we fill was U at position nine. So we XOR the U with the cipher text at the same position, and we got the key, also at the same position. We keep doing that to fill all the empty spaces, and we will have more key bytes. On the ruby tool, I hard coded my guesses like this:

```
key[9] = (ciphers[7][9].ord ^ 'U'.ord).chr
```

We are really closer to the end. Now, it's all about keep guessing and running the ruby tool again. At the end, entire plain text will be revealed.

m0: CAESAR AND VIGENERE ARE CIPHERS THAT CAN BE EASILY BROKEN.

m1:THIS IS THE SECOND STRING THAT WAS ENCRYPTED USING ONE TIME PAD.

m2:ONE TIME PAD WILL PROVIDE A CIPHERTEXT ONLY ATTACK SECURITY.

m3:IN THE CRIPTOGRAPHY CLASSES YOU WILL LEARN HOW TO CONFUSE AND TO DIFFUSE.

m4:TO UNDERSTAND DISCRETE PROBABILITY IS IMPORTANT TO UNDERSTAND CRYPTOGRAPHY.

m5:ENGLISH LETTERS FREQUENCY ALLOWS YOU TO BREAK VIGENERE AND CAESAR CIPHERS.

m6:CRYPTOGRAPHY IS PRESENT IN SEVERAL DIFFERENT TYPES OF APPLICATIONS NOWADAYS.

m7:IF YOU FOUND THE ONE TIME PAD, THEN YOU HAVE FINISHED THIS COURSEWORK.

m8:Every cloud has a silver lining.

IV. Conclusion

Even we considering that one time pad has the perfect secrecy, in this coursework we proved that if we perform an inadequate use of the cipher, is completely possible to break it. We just have to understand a little bit how the XOR operation works, and also explore some hints about it. Most part of the cryptanalysis was completely automated. In fact, just with the result that the tool gave to us, is possible to guess and understand the entire text. I just implemented the guess part to make sure that my guesses are right.

Rafael Rost Software engineer at HP Inc. GitHub account: github.com/rafarost