CSA E0 235: Cryptography (Chalk and Talk-2) January 22, 2016

OTP Implementation Details and Cryptanalysis of Reusing Key

Instructor: Arpita Patra *Submitted by: Sruthi Sekar*

1 Brief Overview Of the talk

We begin by recalling the One Time Pad scheme (Vernam Cipher). The scheme $\Pi = (\mathsf{Gen}(), \mathsf{Enc}(), \mathsf{Dec}())$ is as given below:

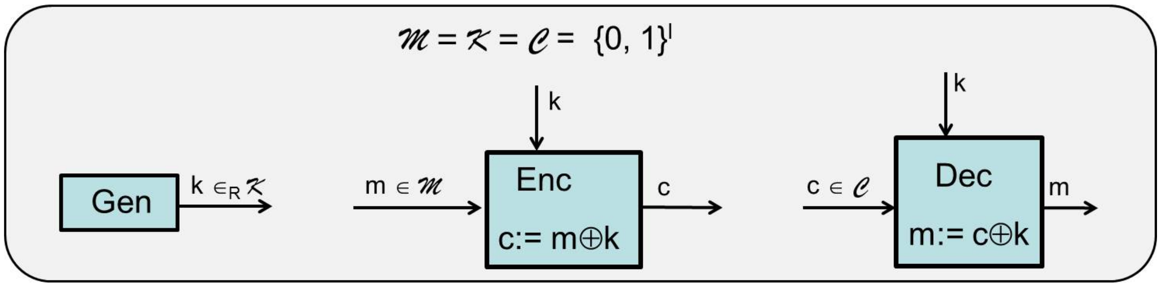


Figure 1: Vernam Cipher

Correctness: $\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m$

Perfect Security: For every distribution over $\mathbf{M}$, every message $m \in \mathbf{M}$ and every cipher text $c \in \mathbf{C}$, the *a priori probability* and the *posteriori probability* are equal, i.e., $Pr[M = m|C = c] = Pr[M = m]$.

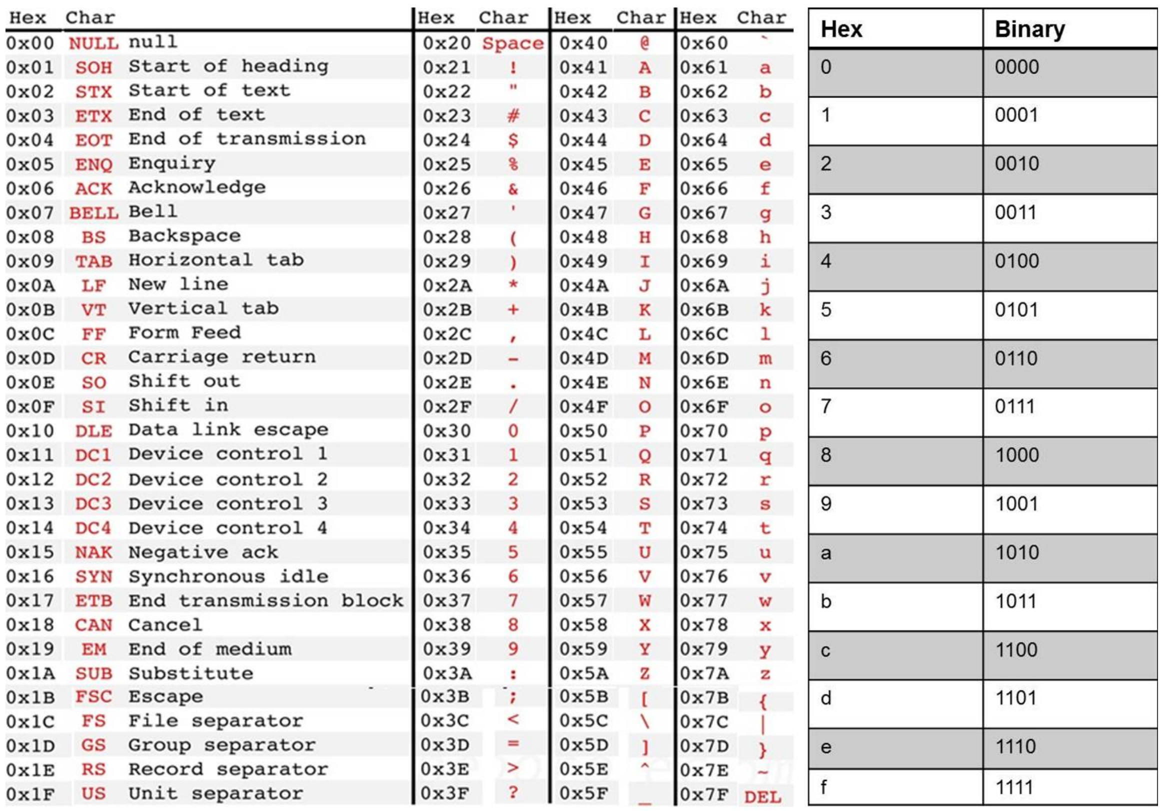
We now see the following:

- **Implementing OTP for plain text in English language:** This involves converting the text into ASCII and performing XOR on the corresponding binary representation of the key and the message. The output is again converted to plaintext.
- **Breaking the OTP when the same key is used to encrypt two different messages:** This is one of the drawbacks of Vernam Cipher, and hence the name One time Pad. This involves frequency analysis using certain redundancies in the English language. There are essentially two steps involved in this, which will be discussed.

2 Implementing OTP

2.1 Hex and ASCII

Before explaining the implementation, we see the standard conversion from string to hexadecimal representation and then from hex to binary. The following table gives the standard hexadecimal representations of English characters, including letters, space, punctuations, numbers and others. Also, the conversion from hex to binary is given.



Hex	Char		Hex	Char	Hex	Char	Hex	Char
0x00	NULL	null	0x20	Space	0x40	@	0x60	`
0x01	SOH	Start of heading	0x21	!	0x41	A	0x61	a
0x02	STX	Start of text	0x22	"	0x42	B	0x62	b
0x03	ETX	End of text	0x23	#	0x43	C	0x63	c
0x04	EOT	End of transmission	0x24	$	0x44	D	0x64	d
0x05	ENQ	Enquiry	0x25	%	0x45	E	0x65	e
0x06	ACK	Acknowledge	0x26	&	0x46	F	0x66	f
0x07	BELL	Bell	0x27	'	0x47	G	0x67	g
0x08	BS	Backspace	0x28	(	0x48	H	0x68	h
0x09	TAB	Horizontal tab	0x29	)	0x49	I	0x69	i
0x0A	LF	New line	0x2A	*	0x4A	J	0x6A	j
0x0B	VT	Vertical tab	0x2B	+	0x4B	K	0x6B	k
0x0C	FF	Form Feed	0x2C	,	0x4C	L	0x6C	l
0x0D	CR	Carriage return	0x2D	-	0x4D	M	0x6D	m
0x0E	SO	Shift out	0x2E	.	0x4E	N	0x6E	n
0x0F	SI	Shift in	0x2F	/	0x4F	O	0x6F	o
0x10	DLE	Data link escape	0x30	0	0x50	P	0x70	p
0x11	DC1	Device control 1	0x31	1	0x51	Q	0x71	q
0x12	DC2	Device control 2	0x32	2	0x52	R	0x72	r
0x13	DC3	Device control 3	0x33	3	0x53	S	0x73	s
0x14	DC4	Device control 4	0x34	4	0x54	T	0x74	t
0x15	NAK	Negative ack	0x35	5	0x55	U	0x75	u
0x16	SYN	Synchronous idle	0x36	6	0x56	V	0x76	v
0x17	ETB	End transmission block	0x37	7	0x57	W	0x77	w
0x18	CAN	Cancel	0x38	8	0x58	X	0x78	x
0x19	EM	End of medium	0x39	9	0x59	Y	0x79	y
0x1A	SUB	Substitute	0x3A	:	0x5A	Z	0x7A	z
0x1B	FSC	Escape	0x3B	;	0x5B	[	0x7B	{
0x1C	FS	File separator	0x3C	<	0x5C	\	0x7C	|
0x1D	GS	Group separator	0x3D	=	0x5D	]	0x7D	}
0x1E	RS	Record separator	0x3E	>	0x5E	^	0x7E	~
0x1F	US	Unit separator	0x3F	?	0x5F	_	0x7F	DEL

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
a	1010
b	1011
c	1100
d	1101
e	1110
f	1111

Figure 2: ASCII Table

When a plain-text english message is chosen to be encrypted, it is first converted in a hexadecimal string to be stored. In order to perform bitwise XOR, we refer to corresponding binary representations shown in Figure 2. For example:
'Asc' in **hex** is: *417363* (See the table)
and in **binary**: *010000010111001101100011*

2.2 Actual implementation

Now, knowing how to get the binary representations, we perform the following steps. These have been implemented in python.

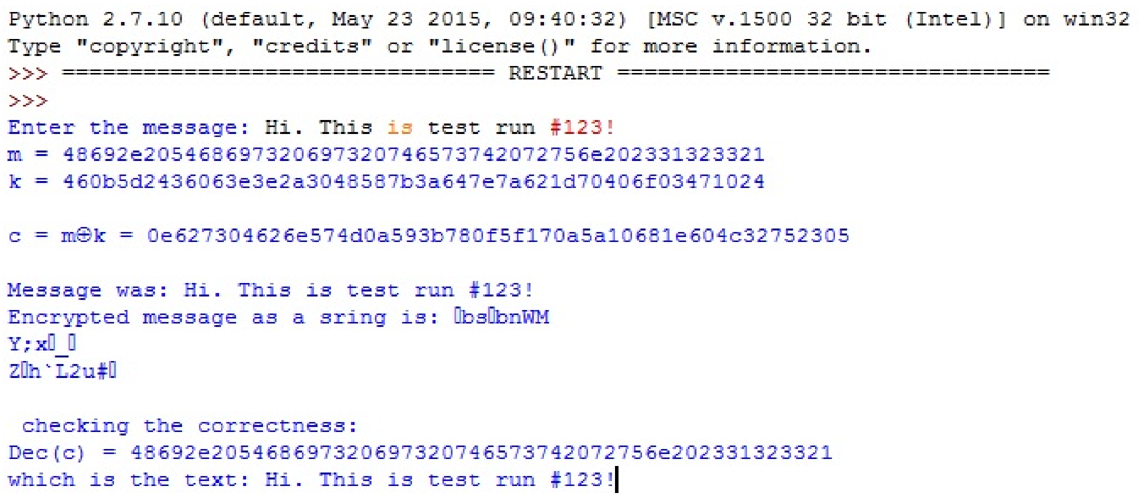
- **Message**: The message is accepted in string format (English text). This is stored as a hexadecimal string to be extracted later.
- **Key Generation**: We need to generate a uniformly random key, with length same as the message length. This is done using a feature in programming called *Crypto libraries*. Although not purely random, but for implementation purposes, it suffices. It is important that every message is encrypted with a unique key. As seen in later section, key reusing breaks the security of the Vernam cipher.
- **Encryption Algorithm**: This takes in two arguments, the message m, and the key

k, and outputs the XOR of the binary representations of these. For example:

$$4865 \oplus 146e = 0100100001100101 \oplus 0001010001101110 = 0101110000001011 = 5c0b$$

- **Decryption Algorithm**: This takes two arguments, the cipher text c, the key k and outputs the XOR of binary representations of c and k and gives m.
- **Verifying Correctness**: In the final step we verify the correctness of the scheme.

A screen shot of a single implementation of the algorithm is shown below in Figure 3.



```
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Enter the message: Hi. This is test run #123!
m = 48692e2054686973206973207465737420727756e202331323321
k = 460b5d2436063e3e2a3048587b3a647e7a621d70406f03471024

c = m⊕k = 0e627304626e574d0a593b780f5f170a5a10681e604c32752305

Message was: Hi. This is test run #123!
Encrypted message as a sring is: bsbnWM
Y;x_
ZhˋL2u#

 checking the correctness:
Dec(c) = 48692e2054686973206973207465737420727756e202331323321
which is the text: Hi. This is test run #123!
```

Figure 3: OTP in Python

3 Cryptanalysis to break Two time pad

The Vernam cipher is perfectly secure but it has several drawbacks:

- The key is as long as the message.
- The length of the message to be communicated needs to be known in advance so that the key of appropriate length is shared a priori to the communication.
- We cannot reuse a key. If two messages are encrypted using the same key, it can be used to gain additional information about the messages sent and hence, the a priori and posteriori probabilities will not remain same. This breaks the perfect security.

$$c_1 = m_1 \oplus k; \ c_2 = m_2 \oplus k$$
$$\implies c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2$$

In the particular scenario where the information being communicated uses messages in English text, we can use several features listed below to actually break the two time pad, i.e., one time pad with a key being re-used, and get the messages which are encrypted using the same key.

- $m_1 \oplus m_2$ reveals the positions where both the messages differ, simply because there xor will be 1 in that position.

- **Characteristics of ASCII Conversions:** There are some observations which can be made in the ASCII conversions. This can be used for a start.

- **Frequency Analysis:** Now, the English language has certain redundancies. A complete analysis of the rates with which single letters, a pair of letters, trigrams and other such combinations occur, has been done and this can be used in predicting chunks of messages. But, this is much more difficult to use on $m_1 \oplus m_2$, as XORing two english language messages tends to smooth out the distributions on the possible values of the XOR. However, if the message is long enough, or if more than two messages have been encrypted using the same key, then the cryptanalysis becomes slightly easier.

Remark : It is important to note that this cryptanalysis is not guaranteed to work all the time. It is a very hard procedure involving a large amount of guesswork and hence might take years of observation of the communication to break the cipher. A practical example of this is mentioned in the next section, the Venona project.

3.1 Properties of ASCII conversions

The first step in the break relies on certain features of the binary representations of letters vs space. Refer to figure 2. We observe the following:

- The binary representation of all letters begin with 01.

- The binary representation of space begins with 00. (So does that of punctuation marks)

- Hence the XOR of two letters gives a 00 in the beginning while the XOR of a letter and a space begins with a 01.
  Remark : We could also get a 01 in the beginning if a letter is XORed with a punctuation mark, but this occurs less frequently than a letter-space combination. So, we can first check using this assumption and proceed.

Remark : This feature can be used to locate those positions in $m_1 \oplus m_2$ which have a letter XORed with a space. But this can be used only if the encryption procedure does not involve removing the spaces between words. In that case, we could analyze letter-punctuation combinations instead.

3.2 Frequency Analysis

This is an involved procedure and there is no definite way of working. We essentially work through by relying on smart guesswork.

- First, we predict a possible string sequence which might appear in one of the messages. For this we might use the frequency ranking of any of the word combinations, but for a start, we use trigram frequency to predict the most possible trigram occurring in either of the two messages. *'the'* is the most common trigram. So this could occur at some position in one of the two messages.

- Assuming this, and combining this with the knowledge of appearance of spaces in either of the two messages (if they exist), we rule out those positions, where 'the' can't occur.
- We then place it at the possible positions, one by one, XOR it with $m_1 \oplus m_2$ at that position and see if the result makes sense. We slide this along, until it does.
- This then shows the position where one of the two messages has 'the'. By expanding the corresponding fragments we have found in the second plaintext, we can work out more of the one time pad, and deduce new fragments in the other message.
- Continue this process back and forth till some sensible messages are deciphered.

3.3 An example to understand the procedure

We begin by taking two ciphertexts c_1 and c_2, which have been encrypted using the same key k. So, we have:

$$c_1 = 3759421c15212b1c1a3566$$

$$c_2 = 0b544b500a7313141a386f$$

$$\implies c_1 \oplus c_2 = 3c0d094c1f523808000d09 = m_1 \oplus m_2$$

So, both ciphertexts have 11 characters.

Step 1 (Predicting the position of space-letter combinations): We write out the first two digits of binary representation of each character.

00.. 00.. 00.. 01.. 00.. 01.. 00.. 00.. 00.. 00.. 00..

↑ ↑

space ⊕ letter (possibly)

So, the fourth and sixth characters are such that one message has a space and the other has a letter. So writing the entire binary representation of space, we get the exact letter at the two positions:

4th : 0100 1100 = 0010 0000 ⊕ ?
So, the letter is: 0110 1100 = **6c** = '*l*'
6th : 0101 0010 = 0010 0000 ⊕ ?
So, the letter is: 0111 0010 = **72** = '**r**'

Step 2(Using trigram frequency): Assume that 'the' occurs in atleast one of the two messages. Since, there is a space in the fourth position, we might guess that 'the' occurs as the first three characters. This guess gives us:

```
  3c 0d 09 4c 1f 52 38 08 00 0d 09     m1⊕ m2
⊕ 74 68 65                             the
-------------------------------------
  48 65 6c                             Hel
```

So, this makes sense, as we have the 4th letter: 'l'. So, one of the messages contains: 'Hello'. And hence:

```
  3c 0d 09 4c 1f 52 38 08 00 0d 09     m1⊕ m2
⊕ 48 65 6c 6c  6f                     Hello
-------------------------------------
  74 68 65 20 70                      the p
```

Now, 6th character in the second message will be 'r'. We guess a 7 letter word starting with 'pr' and check if it gives a meaningful result for the other message:
Guess: 'program', which gives:

```
  3c 0d 09 4c 1f 52 38 08 00 0d 09     m1⊕ m2
⊕ 74 68 65 20 70 72 6f 67 72  61 6d   the program
-------------------------------------
  48 65 6c 6c 6f 20 57 6f  72  6c  64  Hello World
```

This gives us the two messages m_1 and m_2.
Remark : We worked here assuming several things, which might not be our first guess. I have only included those guesses which gave me a sensible result for the plaintext. One could try a different guess and see that it might not give us a meaningful chunk of the other message.

3.4 Venona Project

This is an example where the Vernam cipher was used in practice for communicating delicate messages during war. But a blunder was made when the keys were used repeatedly to encrypt several messages. This led to an ultimate break of the cipher text, revealing the message.

During the initial years of the Cold war, the Venona project was a source of information on Soviet intelligence-gathering activity that was directed at the Western military powers.

Most decipherable messages were transmitted and intercepted between 1942 and 1945. Sometime in 1945, the existence of the Venona program was revealed to the Soviet Union by theUS ArmySIGINTanalyst andcryptologistBill Weisband.These messages were slowly and graduallydecryptedbeginning in 1946 and continuing through 1980, when the Venona program was terminated.

The Soviet company that manufactured the one-time pads produced around *35,000 pages of duplicate key numbers*, as a result of pressures brought about by the German advance on Moscow during World War II.

The duplication, which undermines the security of a one-time system, was discovered and attempts to lessen its impact were made by sending the duplicates to widely-separated users. Despite this, the reuse was detected by cryptologists in the US.

References

[1] Jonathan Katz *https://www.coursera.org/learn/cryptography/home/week/1*. Online course,2016.

[2] Simon Singh *The Code Book*. Thompson Press, 1999.

[3] David Kahn *The Codebreakers*.