Crypto 101: Introduction to Classical Cryptography and Cryptanalysis

Cousin Wu

October 14, 2019

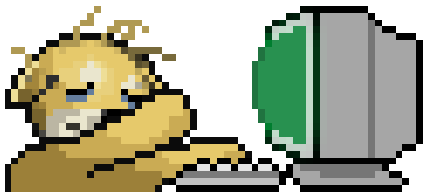
Section 1

Introduction

About the Speaker

No one cares.
I am Cousin Wu
Year 3 UG in Mathematics (PMA) and Computer Science
NOT a winner in PwC Hackaday 2019
Whatever

Interested in math (algebra, geometry, number theory), computer science and cryptography. Feel free to discuss or ask any questions on those.



What is Cryptography?

Cryptography is the study of secure communication.

Example

Encryption is a way to prevent anyone other than the receiver who get hold of the message to get the actual content.

Cryptography emerged thousands of years ago! Julius Caesar used his cipher to protect the message :) (will talk about this later). Those classical cipher could be far from trivial.

Are there any cryptography besides encryption?

Of course yes! E.g. Key exchange, digital signature, cryptography hashes, secret sharing, zero-knowledge proofs...
But we will only cover encryption.Or is it?

Yes. But maybe we can discuss those in the future :)

History (No one cares)

Around the second world war, modern cryptography that made use of mechanical machinery made things more complicated. Enigma machine was used by the Nazi German! The computer science god Alan Turing helped breaking it. AES, one of the most wide used symmetric key encryption scheme, was created in 1997 only! Not much older than you and me!

As you can see, cryptography was, and is still being used, and considered as munitions. (PGP)

Cryptanalysis

Cryptanalysis is the study of breaking cryptography. We can either attack the protocol itself, which are usually difficult, or attack implementations of such protocols (e.g. bad random number generation, bad prime number generation. etc.).

Example (Brute Force)

We could try out all the possible ways to decrypt a message, one of the ways would be the correct way!

This is what we will be doing in CTFs.

Never Implement Your Own Crypto

NEVER implement your own crypto!!!!
NEVER implement your own crypto!!!!
NEVER implement your own crypto!!!!

If something is important, you gotta say it three times.

Why?

- Commonly used ciphers are implemented by others already. (fast and secure enough)
- Need to exactly follow the protocol.
- Side-channel attacks!
- Most ciphers are tested and attacked by thousands of academias before considered safe enough!

Why Crypto in CTF?

- Important in information security
- Any data transfer we do on the internet is protected by crypto!
- Badly implemented crypto appears in real life!
- It is fun! :)

Prerequisites?

Do I need a lot of math?
You need some, but not a lot.
What courses have you taken?

1. Discrete Math? (MATH 2343, COMP 2711, COMP 2711H)
2. Cybersecurity/Cryptography course? (COMP 3632 , COMP 4631, MATH 4632, COMP 5631)
3. Number Theory? (MATH 4141)
4. Courses training math maturity? (COMP 2711/H, MATH 1023, MATH 2033)
5. Abstract Algebra? (MATH 3121/3131/5111/5112)

If you have taken (or are taking) those, then great! If not, don't worry! We will develop the necessary knowledge as we needed them. I would say that discrete math and COMP 4631 will be the most helpful here.

There are some actual requirements though.
You need to use python (with pwntools!). You need to install some scientific computation library and crypto library (**required**): gmpy2, pycrypto. Install with

```
python3 -m pip install pycrypto --user
sudo apt install python-gmpy2
```

Sage, a computer algebra system, would be helpful along the lines. (The size of sage is massive (GB's)! You could use the online one instead.)

Brief Overview of Crypto Training

We will talk about two areas of crypto for the trainings (tentative, depends on request).

- Symmetric Key Cryptography (1 key)
- Public Key Cryptography (2 keys, each for encryption and decryption)

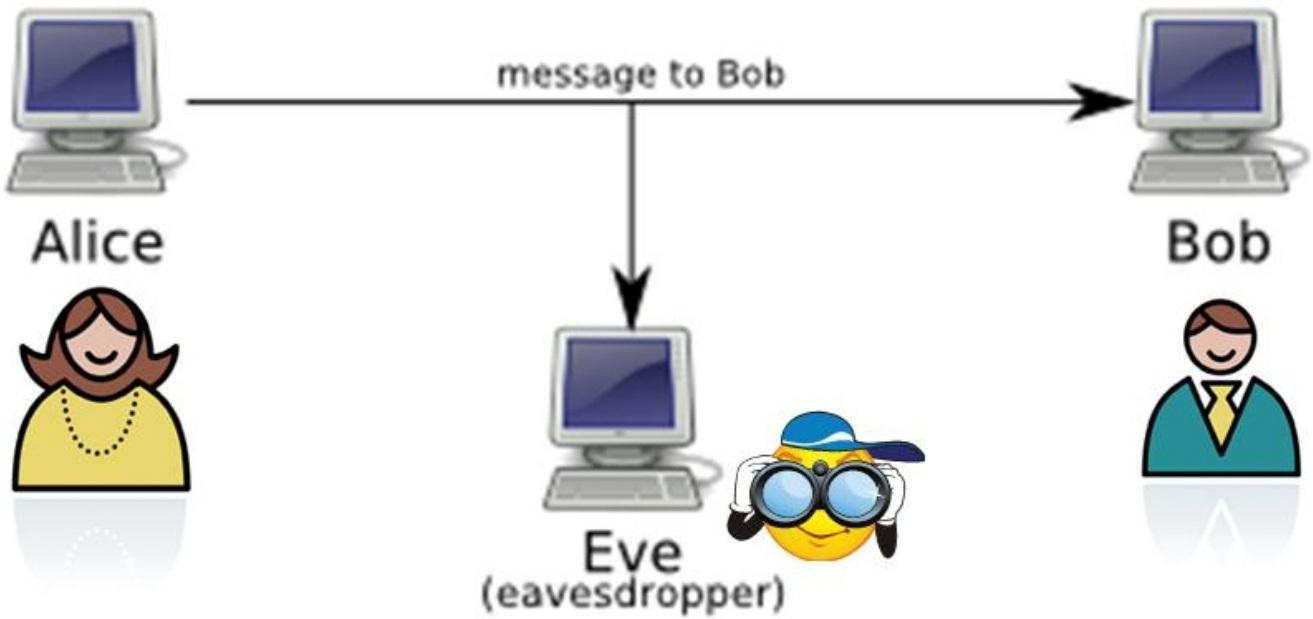
There are also other areas that appears in CTFs, like cryptographic hashes, PRNGs, quantum cryptography, post-quantum cryptography, secret sharing,. . .

Non-Expectation

However, we will **NOT** talk about real world cryptosystem and protocols, such
as PGP, SSL/TLS, IPsec, VPN, SSH,... You will learn those in COMP 4631.

And No, we will **NOT** talk about blockchain in crypto training. (or is it?)

Model



Alice sends messages to Bob through a communication channel, and Eve (us!) can intercept all the messages in the channel. We usually assume/know what encryption/signature/whatever scheme they are using (except for the key of course).

Modern cryptosystems should be secure even if the scheme is known! This is known as the Kerckhoffs's principle. As a corollary, **please do not design a proprietary/secret cryptosystem**.

Figure: A meme.

Encryption

We have already talked about encryption here. But what is encryption?

Definition (Encryption)

Encryption and decryption are functions E_k and $D_{k'}$ bijections with key $k, k\prime$ such that $D_{k'}(E_k(m)) = m$ for all messages m supported by the encryption scheme.
If k' is known, it should be easy to decrypt a message. Conversely, if k' is not known, it should be very difficult (impossible) to decipher. Same goes for encryption and k.

In simpler language, **if you encrypt and then decrypt a message, you should get back the message**.

How to **decipher** the definition of encryption? Lets look at non-examples of encryptions: (treating x as numbers, x can be any real number)

1. $E_k(x) = 0$
2. $E_k(x) = x$
3. $E_k(x) = \sin(x^k)$

Why are these not encryptions?

Encryption vs Encoding

Encoding is a conversion between different formats. Base64, ASCII,...
It is very easy to decode and recognize.

For Encryption, even if the encryption scheme is known, if the key is not known, it should be hard to decipher.

Section 2

Symmetric Key Encryption

Symmetric Key Encryption

An analogy of symmetric key encryption is when we lock things up in a box using a lock. We use the same key to **lock** and **unlock** the box. The catchphrase is that **the key used for encryption and decryption would be the same key**.

Modern Symmetric Ciphers

Definition (Block Cipher)

A **Block Cipher** cuts a long message into several blocks of set length, and operate on each block with the same key.

An example of modern block cipher is AES.

Definition (Stream Cipher)

A **Stream Cipher** uses the (short) key to generate a long key to encrypt long messages.

An example of modern stream cipher is salsa20.

Lets start by learning classical ciphers.

Section 3

Substitution Cipher

Substitution and Permutation Ciphers

Substitution and Permutation cipher are old techniques used for encryption, and they were (quite easily) done by hand, as there were no computers 2000 years ago. That said, it is still being used today for modern ciphers, but much more difficult to do with pen and paper.

Definition (Permutation)

A **permutation** of n elements is a bijection from the set of n elements to itself.

Example (S_3)

Consider the following permutation rule of 3 numbers 1,2,3: 1 goes to 2, 2 goes to 1, and 3 is fixed in place.

Substitution Cipher

A Substitution cipher is where characters of the plaintext are substituted with other characters with some rules.

Example (Simple substitution cipher for S_3)

Consider the following permutation rule of 3 numbers 1,2,3: 1 goes to 2, 2 goes to 1, and 3 is fixed in place. We can define an encryption scheme by applying the permutation to each character in the plaintext. For example, encrypting the word $m = 13231$ gives 23132.

Caesar Cipher

Caesar cipher is an ancient cipher developed 2000 years ago by Julius Caesar. The encryption just moves every character to the right by 3, wrapping back to a if necessary. So A becomes D, B becomes E, and so on, and Z becomes C.

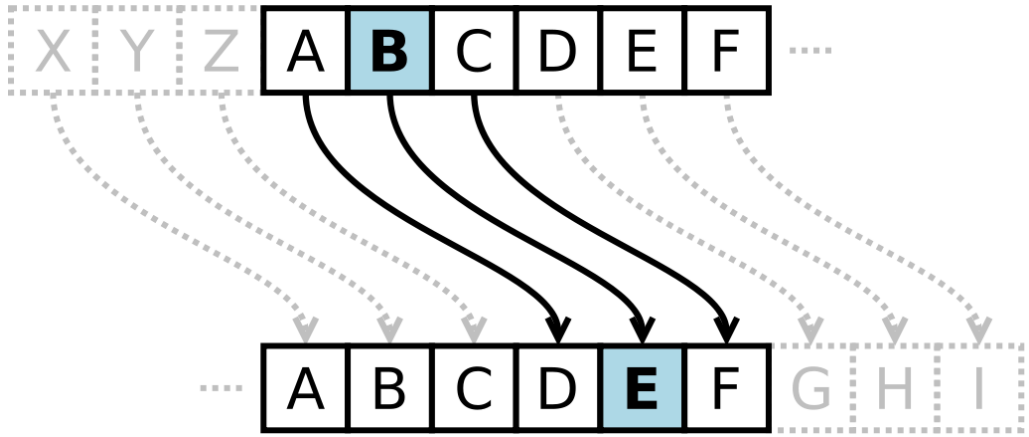


Figure: An illustration of the Caesar Cipher.

Of course it is possible to move character to the right by an offset other than 3. you can use 1, 2, . . . or 25 as the **key** of the cipher ($k = 0$ is stupid).

Formally, if we represent (upper case) English characters as numbers, with 0 as A, 1 as B and et cetera, we can define the encryption and decryption function as

$$E_k(x) = x + k \pmod{26}$$

and

$$D_k(x) = x - k \pmod{26}$$

Exercise

Try decrypt the word "FDQBRXUHDGWKLV" with $k = 3$.

Example (ROT13)

ROT13 is an **encoding** with $k = 13$. If we apply ROT13 twice, what will happen?

Demo

How to break it?

Just try out all the possibilities! There are only 26 possible keys!

dcode

cryptii

Encrypting with Caesar Cipher

```
def caesar(m, k):
    m_number = []
    m = m.upper()
    # Converting string to base26
    m_number = [ord(char) - ord('A') for char in m]
    rotated = [(i + k) % 26 for i in m_number]
    # converting base26 back to string
    c_number = [chr(i + ord('A')) for i in rotated]
    result_string = "".join(c_number)
    return result_string


C = "HKUSTFIREBIRD"
print(caesar(C, 18))
```

Sample Brute Force Code for Caesar Cipher

```
def caesar_brute_force(c):
    c_number = []
    c = c.upper()
    # Converting string to base26
    c_number = [ord(char) - ord('A') for char in c]
    for k in range(26):
        rotated = [(i + k) % 26 for i in c_number]
        # converting base26 back to string
        m_number = [chr(i + ord('A')) for i in rotated]
        result_string = "".join(m_number)
        print("{}:\t{}".format(k, result_string))


C = "QEBNRFZHYOLTKCLUGRJMPLSBOQEBIXWVALD"
caesar_brute_force(C)
```

Remark

Of course it is possible to use character sets other than the English alphabet. e.g. alphanumeric characters, ASCII, etc.
Basically you need to convert your characters to numbers first by **encoding**, which here just means assigning each character with an integer between 0 and the size of your alphabet -1.

The encoding matters! Different encoding gives different encryption result! (see HW)

Note that caesar cipher can be viewed as a block cipher with block size of 1. Same for the coming substitution ciphers as well.

Affine Cipher

Affine cipher is also an substitution cipher similar to caesar cipher, but the encryption and decryption is a bit more complicated. If we **encode** each character from the character set as 0,1,. . . , n, and define the key $k = (a, b)$ where a and n do not share common factors other than 1, then

$$E_k(m) = ax + b \pmod{n}.$$

Exercise

What is the decryption function?
The answer should be trivial, so it is left as exercise to the reader.

The reason why a needs to not share common factors with n can be found in the supplementary notes.

Weakness

If n is large, then the possibilities of a and b are huge! Then it may be infeasible to brute force. However, if n is small, e.g. 26, then there are only $12 \times 26 = 312$ possible keys. So it is still possible to brute force that. A more deadly weakness of affine cipher is that it is vulnerable to a known-plaintext attack.

Definition (Known-plaintext Attack)

A **known-plaintext attack** (KPA) is an attack model where the attackers have knowledge of a ciphertext and its decrypted plaintext.

How to recover the key from plaintext and ciphertext?

Claim

Suppose we have the encrypted and plaintext version of **2** characters. Then we can cover the key (a, b).

KPA for Affine Cipher

Proof.

We get

$$c_1 \equiv ax_1 + b \pmod{n} \tag{1}$$

$$c_2 \equiv ax_2 + b \pmod{n} \tag{2}$$

(1) − (2) yields

$$c_1 - c_2 \equiv a(x_1 - x_2) \pmod{n}$$

. Then we can solve for a. With a, we can solve b as well. □

Note that this only works if $x_1 - x_2$ do not share any common factors with n other than 1. This requirement should look trivial to those who study any elementary number theory.

Homework (Affine)

```
from secret import flag
from math import gcd
from random import randint
from string import ascii_letters, digits, punctuation

CHARSET = ascii_letters + digits + punctuation
n = len(CHARSET)


def affine(m, a, b):
    m_numbered = (CHARSET.find(i) for i in m)
    c_numbered = ((a * x + b) % n for x in m_numbered)
    return "".join(CHARSET[k] for k in c_numbered)
```

Homework (Affine) (Cont'd)

```
if __name__ == "__main__":
    a = 0
    b = randint(0, n - 1)
    while gcd(a, n) != 1:
        a = randint(2, n)
    secret = affine(flag, a, b)
    print("Good day! Please find the secret below")
    print(secret)
    print("kthxbye")
    print(affine("haha", a, b))
```

Extra Question

The below problem is too hard. If you have learnt (or are learning) linear algebra, then attempting this would be a good exercise.

Exercise

Suppose the n in affine cipher is unknown.
Proof that if n is a prime, then given 3 plaintext-ciphertext pair, one can recover n (disregarding the running time), and thus break the cipher.
Hint: Let p be a prime, $\mathbf{A}$ be a nxn matrix with integer entries, and $\vec{x} \in \mathbb{Z}^n$. If $\mathbf{A}\vec{x} \equiv \vec{0} \pmod{p}$ has solutions other than $\vec{x} \equiv \vec{0} \pmod{p}$, then $\det \mathbf{A} \equiv 0 \pmod{p}$.

Simple Substitution Cipher

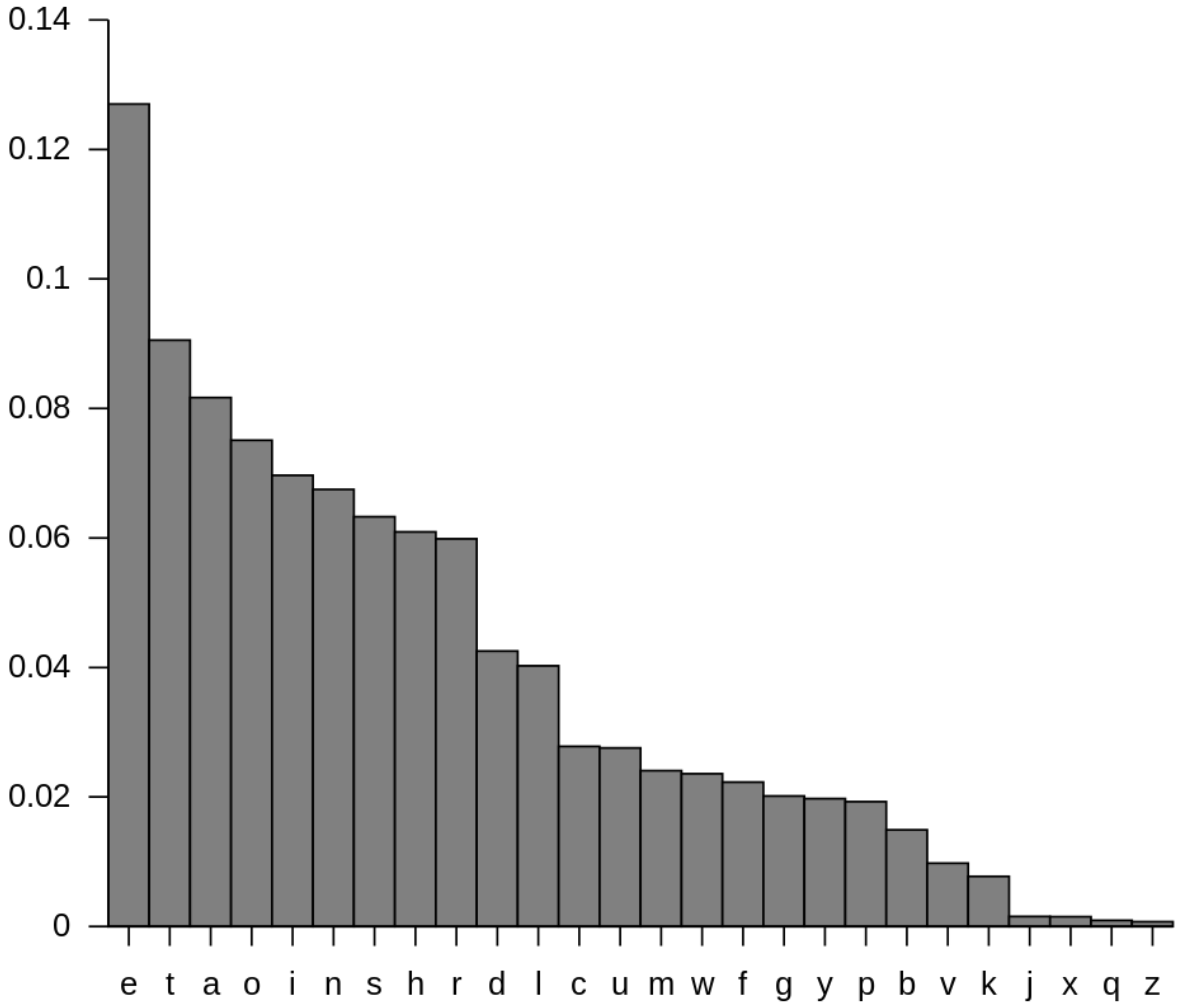
So far we saw substitution rules defined with a linear equation mod n. How about arbitrary permutations?

q11 from firebird.ozetta.net
https://www.thephoenixsociety.org/puzzles/puzzles.htm

There are 26! possible permutations of 26 characters! We cannot just brute force this. What do we do?

Frequency Analysis

The distribution of English characters among common English texts are not uniform nor random.



R	40×	12.35%
O	39×	12.04%
I	30×	9.26%
E	26×	8.02%
K	21×	6.48%
P	20×	6.17%
D	17×	5.25%
G	16×	4.94%
F	15×	4.63%
W	14×	4.32%
X	11×	3.4%
N	11×	3.4%
V	10×	3.09%
Z	9×	2.78%
Q	8×	2.47%
Y	7×	2.16%
L	7×	2.16%
U	7×	2.16%
H	6×	1.85%
A	5×	1.54%
J	3×	0.93%
B	2×	0.62%

The same goes to digraphs, which is a pair of characters (e.g. th), and trigraph (guess what that means :)). However, doing it by hand is painful. So there is a tool for it :) quipquip

Note that if the length of ciphertext is short, then this could not be done :(

This shows that the simple substitution cipher is not secure against a ciphertext-only attack.

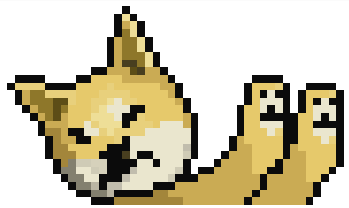
Definition (Ciphertext-only attack)

A **ciphertext-only attack** (COA) is an attack model where the attackers have knowledge of some ciphertext only.

Remark

Since the affine cipher is also just a special case of the simple substitution cipher, affine cipher is also vulnerable to frequency analysis!

Actually the same goes to Caesar cipher. But why though?



Vigenère[1] Cipher

So far we have seen substitution ciphers with block size 1. Now we will see a stream cipher. Now the key becomes a string of characters, e.g. English again, say $k = k_1 \cdots k_6 =$"CRYPTO". Here the key length is 6. By converting the characters into numbers, we can define the following encryption:

$$E_k(x_i) = x_i + \bar{k}_i \pmod{n}$$

However, our key length is only 6, and our message can get much longer than that! So if x_i is larger than 6, we need to repeat the key. Thus we have $\bar{k}_i = k_{(i \bmod m)}$, where m is the key length.

[1] I won't attempt to pronounce this word.

Example

As an example, lets try to encrypt "CRYOTOISFUN" using the key "CRYPTO".

Plaintext	C	R	Y	P	T	O	I	S	F	U	N
base26	2	17	24	15	19	14	8	18	5	20	13
key	C	R	Y	P	T	O	C	R	Y	P	T
Key in base26	2	17	24	15	19	14	2	17	24	15	19
$x_i + k_i \pmod{26}$	4	9	22	4	12	2	10	9	3	9	6
Ciphertext	E	J	W	E	M	C	K	J	D	J	G

How to break this?

Note that if we group every m-th characters together, each group are encrypted using the same character, meaning that the encryption reduces to a Caesar cipher! However, if the key length is not known, then this method is not very helpful. So one possible decryption method is as follows:

1. guess the key length
2. frequency analysis on respective key

Another possible approach is to guess the key. If the key is in English, then it is probably a valid English word! Then we can try to do a dictionary attack. As a corollary, never use an English word as the encryption key for this cipher.

Exercise

Try to implement the Vigenère cipher in python.
You may find the library **itertools** or a library function **zip** helpful.

Section 4

Transposition Cipher

Transposition (Permutation) Cipher

For substution cipher, we permute the character set. For **transposition cipher**, we permute the plaintext directly. A simple example would be to first fix the block size, say n, and pick a key k to be an arbitrary permutation of n characters. Then the encryption for each block is just permuting the characters in each block according to the rule.

Example (Transposition cipher for S_3)

Consider the following permutation rule of 3 numbers 1,2,3: 1 goes to 2, 2 goes to 1, and 3 is fixed in place. Using the example encryption scheme given above, encrypting the word $m = 13231$ gives 31213.

Columnar Cipher

Another pen-and-paper cipher.

Write the plaintext in rows of n, then permute the columns. Write the words column by column to get the ciphertext. Do it again with different permutations of columns to get a double columnar cipher.

http://rumkin.com/tools/cipher/coltrans.php

How to break this?

Brute force? Need to guess the key length and the key! thats $\sum_{i=1}^{n} i! > n!$ guesses, which is a lot.
Dictionary attack on the key?
Anagrams? (Nag a ram?)

Question

Exercise

Suppose you are given a string of English characters as ciphertext. How to tell what type of cipher, substitution cipher or transposition cipher, the string is encrypted with?

HW

Section 5

Introduction to Modern Block Ciphers

Modern Block Ciphers

Before the end, we shall introduce some precursors for modern block ciphers.

Bitwise operators

Consider a boolean operator, e.g. the **and** operator AND : $\{0,1\} \times \{0,1\} \to \{0,1\}$. We can extend the definition of such an operator to a binary string by defining

$$\text{AND}^*(a_1 a_2 \cdots a_n, b_1 b_2 \cdots b_n) = \text{AND}(a_1, a_2)\,\text{AND}(b_1, b_2) \cdots \text{AND}(a_n, b_n)$$

Where a_i and b_i are bits, and ab means concatenation of a and b.

Simply put, we operate on two binary strings bit by bit using the boolean operator. That give us a bitwise operator. Note that **the two operands need to have the same length**.

Example

1101 and 1011 = 1001.

XOR

To learn modern block ciphers, we need to learn the very important bitwise operation xor, which stands for **exclusive or**. This is often denoted as $\oplus$ The action of xor are described by the following truth table:

$\oplus$	0	1
0	0	1
1	1	0

Alternatively, we can consider the xor operation acting on 2 binary string of equal size as an addition without carrying.

Why is this called the **exclusive** or? Compare this with the bitwise or operation, and note that 1 or 1 evaluates to 1, but $1 \oplus 1 = 0$.

Example

In python, we can do xor by the $\wedge$ operator. For example, we can write $x \wedge y$ to do $x \oplus y$.

Remark

Not to confuse xor $\oplus$ with the direct sum $\oplus$ in linear algebra, and not to confuse the xor in python $\wedge$ with the power symbol we usually write as $\wedge$ or the wedge product $\wedge$.

Example

$100110 \oplus 001100 = 101010$.

$$\begin{array}{cccccc} & 1 & 0 & 0 & 1 & 1 & 0 \\ \oplus & 0 & 0 & 1 & 1 & 0 & 0 \\ \hline = & 1 & 0 & 1 & 0 & 1 & 0 \end{array}$$

Properties of xor

1. (Commutativity) $a \oplus b = b \oplus a$
2. (Associativity) $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
3. (Zero) $a \oplus 0 = 0 \oplus a = a$
4. (Self-inverse) $a \oplus a = 0$

The last one is especially import for doing calculation on reversing xor operations.

Exercise

Proof all the properties above. (Hint: truth table)

Exercise

Show that $(a \oplus b) \oplus a = b$ for boolean variable a and b.

XOR encryption

Suppose we have a message encoded with binary (e.g. hex). Then one way to encrypt a message is to apply xor with a key. Suppose we have a m bit long message x and a n bit number k as a key, and $m = n$, then we can define the following encryption:

$$E_k(x) = x \oplus k$$

Without the key, we cannot recover the message.

Exercise

Write down the decryption function. Check that if you encrypt then decrypt, you get back the message.

Remark

This shows that $E_k(x)$ is a bijection. (This means that E_k permutes all strings of length n.)

Cryptanalysis

Claim

XOR encryption is vulnerable to a known-plaintext attack.

How?

Let x, k and c be the plaintext, the key and the ciphertext respectively. We have $c = x \oplus k$. Suppose we know x and c. Then

$$\begin{aligned} c \oplus x =& (x \oplus k) \oplus x \\ =& k \end{aligned}$$

As a corollary, we should not reuse keys!

One-time Pad

The idea that we must not reuse keys for the xor encryption give rise to the **one-time pad** (Remark: this is not just for xor encryption).

Each time we encrypt something, we can use a brand new key, where the keys are exchanged to the other person through some side channel (e.g. a code-book).

Suppose the code-book is not known to anyone else, the one-time pad achieves **perfect secrecy**!!! Which means that the ciphertext gives **no** information about the content of plaintext.

Many-time Pad

What if a key is used to xor several plaintexts but we do not have any corresponding plaintext?

Consider two plaintext x_1, x_2 and their corresponding ciphertext c_1, c_2. We have that $c_1 = x_1 \oplus k$ and $c_2 = x_2 \oplus k$.

If we do $c_1 \oplus c_2$, then we get $x_1 \oplus x_2$ (the derivation is left as exercise). Although we do not recover the key nor the message, we do have **partial** information about the plaintext. With another observation, one can attempt to recover parts of the key.

Exercise

Suppose we have x and y with $0x41 \leq x \leq 0x5a$ and $0x61 \leq y \leq 0x7a$. What is the result of $0x20 \oplus x$ and $0x20 \oplus y$?

Substitution-Permutation Network

It would seem that the substitution and permutation ciphers are simple, and it is. Some of them is easily broken with today's computation power. But what if we combine them together?

A Substitution-Permutation Network (SPN) make use of both substitution cipher (S-box) and permutation cipher (P-box). By applying several rounds of S-box and P-box and combining with some round key generated from a key each round, we get a hard-to-break cipher.

If one bit of the plaintext is changed, then it will change several bits of the output of the S and P-boxes, and so on. What we get in the end looks completely random. At the same time, if we change one bit of the key, then it will change the input of the S and P-boxes as well.

SPN is used in the AES encryption scheme, one of the most widely used modern symmetric key cipher currently.

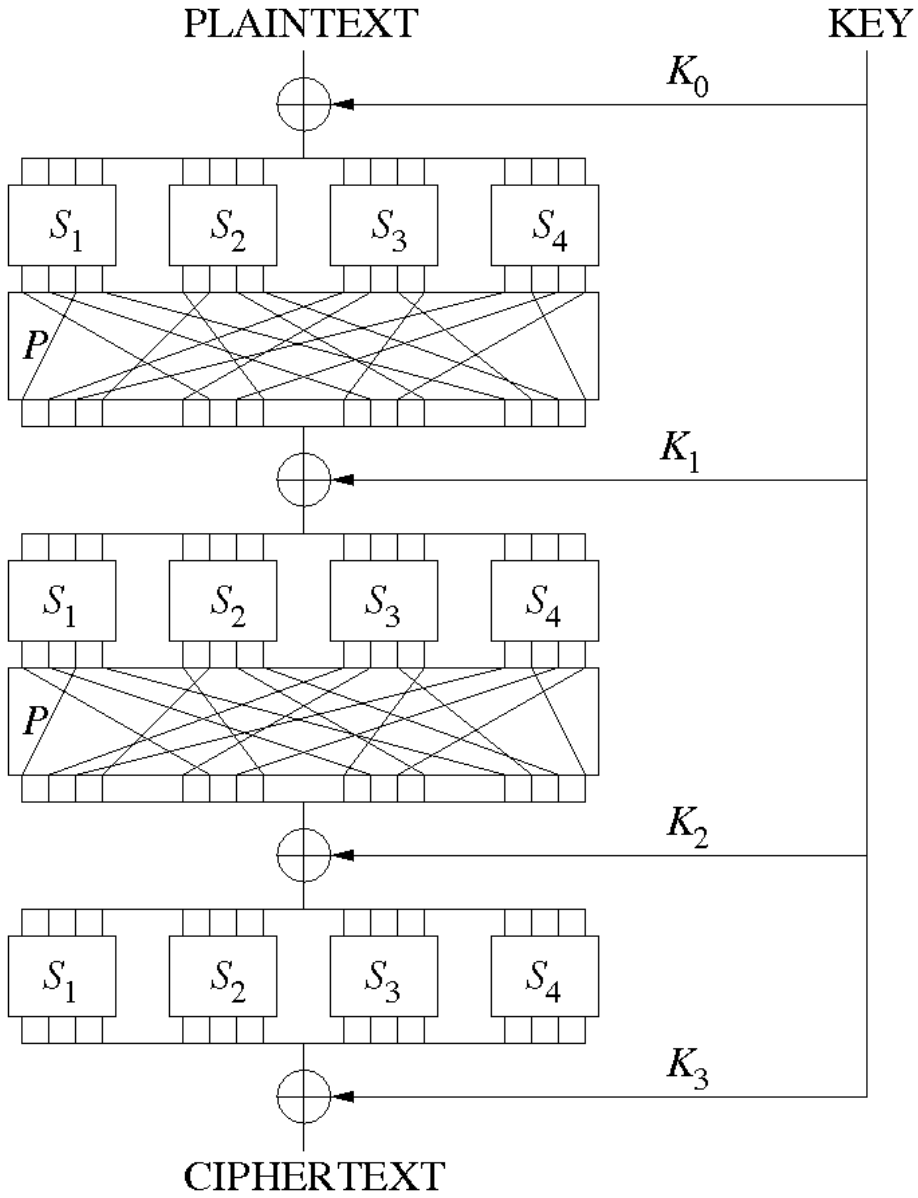


Figure: An example of a SPN

Section 6

Supplementary Notes

Supplementary Notes

The supplementary notes aims at providing necessary mathematical background for understanding the contents in the main notes.
The proof of theorems stated here will be omitted. Interested readers should ask the speaker for reference or look for suitable books or lecture notes to learn more. This is vital in actually **understanding** why things work, and not just memorising and working with definitions and theorems without knowing the inner workings.

Here we will introduce modular arithmetic. If the reader have taken any course on elementary theory, e.g. COMP 2711/H, MATH 2343, MATH 4141, then the reader can skip that section.

Modular Arithmetic

Modular arithmetic is an import tool for cryptography. Simply speaking, we want to ask for the remainder of some number dividing other elements. For example, the remainder of 14 dividing 5 is 4. In other words,

$$14 = 2 \times 5 + 4$$

We introduce a notation:

$$14 \bmod 5 = 4,$$

read as 14 mod 5 equals 4. Alternatively, we have another notation:

Definition (modulo)

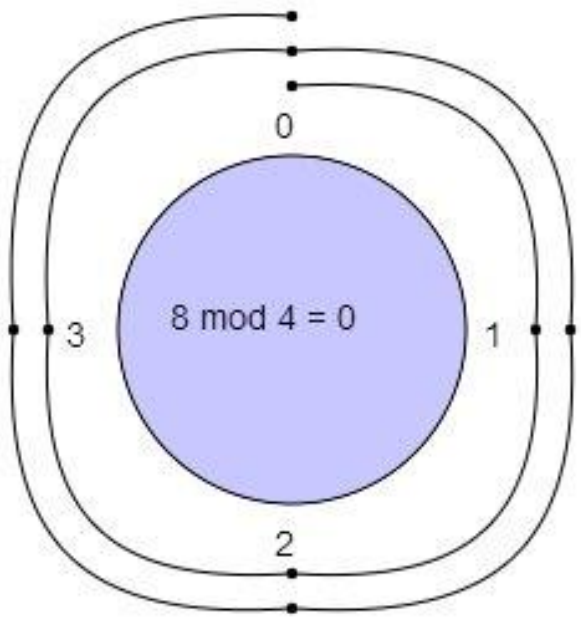
If the remainder of two numbers a, b dividing by another non-zero number n are equal, we write

$$a \equiv b \pmod{n}$$

We read this as a is congruent to b mod (or modulo) n.

In python, we can do modular arithmetic by the percent operator %. For example,

$$14\%5 == 4.$$



Example

$14 \equiv 4 \pmod 5$.
$4 \equiv -1 \pmod 5$.
$k \equiv 1 \pmod 2$ whenever k is odd.
15 : 00 and 3 : 00 looks the same on a clock. This is because $15 \equiv 3 \pmod{12}$.

Properties

Let a, b, c be any integers. Then

- (Reflexivity) $a \equiv a \pmod{n}$
- (Symmetry) $a \equiv b \pmod{n}$ means $b \equiv a \pmod{n}$ as well.
- (Transitivity) $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ implies

$$a \equiv c \pmod{n}$$

If $a \equiv b \pmod{n}$, then

- $a + c \equiv b + c \pmod{n}$
- $a - c \equiv b - c \pmod{n}$
- $ac = bc \pmod{n}$

Multiplicative inverse

Definition (GCD)

We say the **greatest common divisor** (gcd) of two numbers a and b to be c, denoted $\gcd(a, b) = c$, if

1. c is a factor of both a and b
2. c is the largest such factor.

Theorem

If $ac = bc \pmod{n}$ *and* $\gcd(c, n) = 1$, *then* $a = b \pmod{n}$.

This theorem is important because it motivates us to define the inverse (kind of division) inside this modulo system (or residue system in number theory terms).

Definition (Multiplicative Inverse)

If $a \times b \equiv 1 \pmod{n}$ where a, b, n are integers, n positive, then we say that a has a multiplicative inverse b mod n, denoted $a^{-1} = b$.

Theorem

Multiplicative inverse of a exists mod n if and only if $\gcd(a, n) = 1$.

Given an equation $ax \equiv b \pmod{n}$, if the multiplicative inverse of a exists, then we can move a to the other side to obtain $x \equiv a^{-1}b \pmod{n}$. The requirement is that $\gcd(a, n) = 1$.

Actually if $\gcd(a, n) \neq 1$, solution may still exist subject to condition on b. This becomes trivial once the reader learn about Euclidean algorithm and Bezout's identity. However, these two results will be delayed until we discuss the RSA encryption scheme.

Example

Solve $2x \equiv 5 \pmod 7$.
First note that $\gcd(2, 7) = 1$ so solution must exist (uniquely in mod 7).
Then we can calculate to see that $2 \times 4 = 8 \equiv 1 \pmod 7$ so the multiplicative inverse of 2 mod 7 is 4. Finally we get

$$\begin{aligned} 4 \times 2x &\equiv 4 \times 5 && \pmod 7 \\ 1 \times x &\equiv 20 && \pmod 7 \\ x &\equiv 6 && \pmod 7 \end{aligned}$$