UNIVERSITY of DEBRECEN

Dr. József Sütő
Department of IT Systems and Networks

Embedded Systems Laboratory

Lab 3: The concepts of symmetric and public key cryptography

Project overview

In this laboratory you will learn about the concepts of the symmetric and the public key cryptography. Those concepts are important in several embedded systems application where you have to hide information from possible attackers.

The following topics will be covered in this lab:

- Public and private keys
- Caesar cipher
- One-time pad
- Factorization
- Elapsed time measurement

Technical requirements

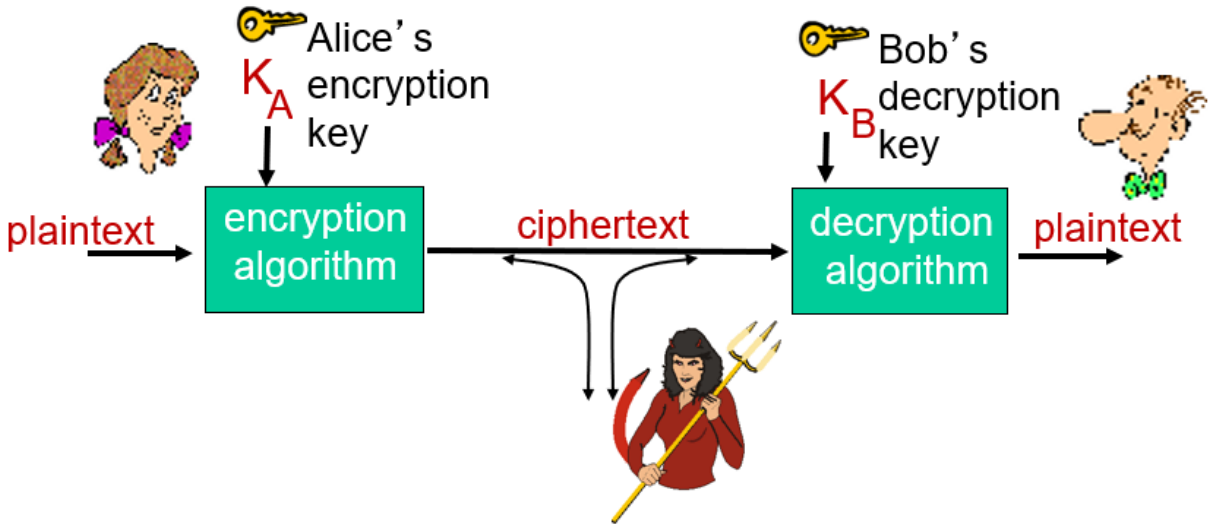
The following components are required to complete this lab:

- Raspberry Pi 3 Model B+ (MicroSD card, power supply, keyboard, mouse)

1. Principles of cryptography

Cryptographic techniques allow a sender to hide data in order to the intruder does not be able to extract useful information from the intercepted data. The receiver, must be able to recover the original data from the encrypted data. For a better understanding we will introduce Alice and Bob, two people who want to communicate "securely". Those names are well-known fixtures in the security community, perhaps because their names are more fun than a generic entity named like "A" or "B". Alice wants only Bob to be able to understand a message that she has sent, even though they *are* communicating over an insecure medium where an intruder (Trudy) may intercept whatever is transmitted from Alice to Bob. Alice provides a **key**,

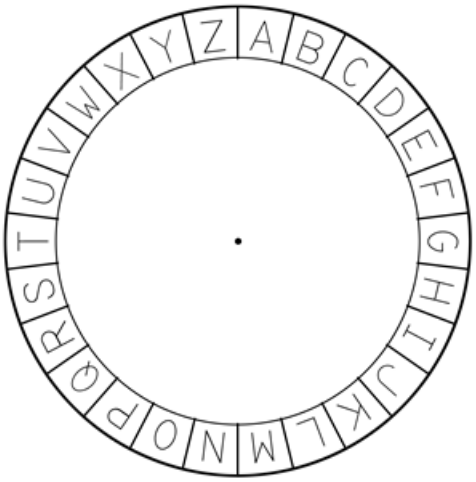
K_A, a string of numbers or characters, as input to the encryption algorithm. The encryption algorithm takes the key and the plaintext message, ***m,*** as input and produces ciphertext as output ($K_A(m)$ refers to the ciphertext). Similarly, Bob will provide a key, K_B, to the **decryption algorithm** that takes the ciphertext and Bob's key as input and produces the original plaintext as output. In **symmetric key systems**, Alice's and Bob's keys are identical and are secret.



2. Caesar cipher

A **cipher** is a type of secret code, where you swap letters of the alphabet around so that no-one can read your message. Caesar cipher is one of the oldest and most famous technique which is named after Julius Caesar. In this method, a **numerical key** is used to **encrypt** a message and the receiver uses the **same key to decrypt** the message. The key needs to be agreed on by both sides beforehand and must be kept secret. An encrypted message using Caesar cipher can be easily decrypted without knowing the key with **brute-force attack** – try all possible keys (25) while you find a correct one.

Let's look at an example to hide the *hello* word (hiding a word is called **encryption!**) using the Caesar cipher. At first, we draw the alphabet in a cycle:



Thereafter, we need a secret key. Suppose that it is 3. The letters of the alphabet are transposed forwards by the number of places given by the key. In the case of 3, the letter A becomes D, the letter B becomes E, and so on. The alphabet is considered to wrap around, so the letter Z would become C. Consequently, the hello word will be encrypted as *khoor*.

3. Caesar cipher - encryption

To encrypt a character, we should add the key to the position of the character in the alphabet. Note that, the alphabet arrangement is a cycle. Thus, according to the new position we turn around the whole alphabet one or more times. The below program code shows, an example to the Caesar encryption.

3.1. Define the header of the Caesar encryption function and an alphabet as a string variable where each letter can be accessed via its index:

def caesar(msg, key):

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

3.2. Encrypted message will be stored in another string:

```
new_message = ""
```

3.3. Add a for loop to the function where a variable goes through all characters in the message. Inside the loop, characters are encrypted one by one. Encrypted characters are added to the encrypted message string.

```
for c in msg:
    position = alphabet.find(c)
    new_position = (position + key) % len(alphabet)
    new_character = alphabet[new_position]
    new_message += new_character
return new_message
```

3.4. The main part of the code is an infinite loop, where we ask the user to enter a message which will be encrypted. This loop will be terminated if the message is '**q**':

```
key = 3
while True:
    message = input("Please enter a message: ")
    if message == 'q': break
    encry_message = caesar(message, key)
```

3.5. Print out encrypted message:

```
print("Encrypted message: ", encry_message)
```

Tasks to be solved alone:

3.A. Modify your program, so that the program be able to handle upper case letters!

4. Caesar cipher - decryption

To decrypt a character, we should subtract the key from the position of the encrypted character. Again, note that, the alphabet arrangement is a cycle. A decrypted position can be a **negative number!** Fortunately, we can handle it easily with the modulo operation.

4.1. Actually, the same function can be used for encryption and decryption because the only difference between the two operations is the sign of the key.

encry_message = caesar(message, key)

decry_message = caesar(encry_message, -key)

4.2. Extend the code in section 3 to perform encryption and decryption as can be seen in the following example output:

```
Please enter message: hello
Encrypted message:  khoor
Decrypted message:  hello
```

5. One-time pad

In the next exercise, you will learn how to generate secret messages using the one-time pad (OTP) technique. When using an OTP, a string of random numbers is generated and shared between Alice and Bob. Each letter of the message is then shifted by the corresponding number in the OTP, so each letter has its own individual key!

5.1. Import the randint() method from random module

from random import randint

5.2. Define the alphabet that will be used.

ALPHABET = 'abcdefghijklmnopqrstuvwxyz'

5.3. Create a function to generate OTP. It has one parameter, the number of characters in the message that will be encrypted. The generated key (sequence of random numbers) will be stored in a text file line by line. Since our alphabet consist of 26 characters, random numbers are between 0 and 25.

def generate_otp(characters):

```
        with open("otp.txt", "w") as f:
            for i in range(characters):
                f.write(str(randint(0,26)) + "\n")
```

5.4. We need another function that load the key from file. In the next code, the **sliptlines()** breaks the content of the text file into lines and eliminates the '**\n**' character from the end of the lines.

```
    def load_otp():
        with open("otp.txt", "r") as f:
            contents = f.read().splitlines()
        return contents
```

5.5. The next function will perform the encryption. It has two parameters the message and the key. Here, we have a loop that goes through all characters in the message. If the character is out of the alphabet, it will be added to the ciphertext as it is. If the character is in the alphabet, it will be shifted by the appropriate value of the key. Since, the shifted character position can be greater than 25, we need to use the modulo operation.

```
    def encrypt(message, key):
        ciphertext = ''
        for (position, character) in enumerate(message):
            if character not in ALPHABET:
                ciphertext += character
            else:
                encrypted = (ALPHABET.index(character) +
int(key[position])) % len(ALPHABET)
                ciphertext += ALPHABET[encrypted]
        return ciphertext
```

5.6. The decrypt() function is the same as the encrypt() but the key values are negative.

5.7. Read a message from the user and use the above defined functions to encrypt and decrypt the message. Print out the encrypted and decrypted versions of the original message. (task to be solved alone)

6. Public key cryptography

A public key cryptographic system consists of a public key and a matched but non-identical pair of private keys. The **private keys** are known only to the participants, and each participant holds one of them. The **public key** is created using the matched pair of private keys, and can be known by anyone. Using public key cryptography is like signing a message with your own **digital signature**.

How it works:

- Alice and Bob agree to use a public key cryptographic system and generate one private key each (A and B), as well as a public key (AB) which is created by multiplying A and B (A * B).
- Bob sends Alice the public key AB.
- Alice receives the public key AB, and she knows this public key corresponds to her private key A. Alice encrypts her message with her private key A and sends it to Bob.
- Bob receives Alice's encrypted message. He has the public key AB and his private key B. Bob knows that AB = A * B, so he can work out A (by calculating A = AB / B) and decrypt the message from Alice.

The **public key** can be any number which meets the following criteria:

- It is chosen using a **random** source of information so that it is unpredictable.
- It is the **product of two numbers**, A * B = AB, and A and B are both **prime numbers** (each only divisible by itself and 1). A public key has to be a number with only two factors, meaning dividing the number by any number besides the two factors will leave a remainder. Since two chosen factors are prime numbers, they themselves have no factors. This guarantees that there is exactly one solution to the problem of finding the public key.
- This product **AB is a large number** and therefore has many digits.
- A and B are used as **private keys** and are the only factors of the public key AB.

We can see how this works in practice as follows:

- Suppose we ignore the requirement for the public key to be a large number for now and use small randomly chosen prime numbers, **A=2 and B=5**. This makes the public key **AB = 2 * 5 = 10**. It is easy to work out that A=2 and B=5 are the only possible factors of 10.
- If we use A=2 and B=5 as the private keys, AB = 10 becomes the public key. Let's assume Alice uses A=2 as her private key and Bob uses B=5.
- Now imagine an attacker intercepts the message sent from Bob to Alice. The attacker can find out that the public key was 10 because it will need to be **sent along with the encrypted message**, and the attacker may even know that the private keys are the factors of the public key.
- Therefore, from the attacker's point of view, to break the cryptography they will need to find A and B by finding the factors of the public key

Because we have chosen small value private keys, A=2 and B=5, in this example, it is very easy for the attacker to figure out what these private keys are. All they have to do is multiply all possible values of A and B and see which multiplication results in the value 10. However, if the public key was a larger number, it would be considerably more difficult to work out the **factors** (hard math problem).

7. Choosing public key to use with the Caesar cipher

The Caesar Cipher has 25 possible keys which represent the number of places each letter of the alphabet is transposed to generate the cipher text. Of these, the following numbers are prime (only divisible by itself and 1):

- [2,5]
- [7,11]
- [13,17]
- [19,23]

We have put these into pairs to illustrate 4 possible private key combinations which could be used with the Caesar Cipher. Supposing we chose the first pair in this list, we would calculate the public key by multiplying the two primes together. The public key for this pair is 10. The two private keys of 2 and 5 are held by Alice and Bob. Alice can encrypt a message with her private key, and send the public key with the message to Bob. Bob then uses the combination of the public key and his private key to work out the key Alice used to encrypt her message. Bob can then decrypt the message. An example code to the public key version of the Caesar cipher can be seen below.

7.1. Create an encryption function with a private key:

```
def caesar_encryption(msg):
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    privare_key = 19
    encry_message = ""

    for c in msg:
        position = alphabet.find(c)
        new_position = (position + privare_key) % len(alphabet)
        new_character = alphabet[new_position]
        encry_message += new_character
    return encry_message
```

7.2. Create an decryption function which can handle public key:

```
def caesar_decryption(msg, public_key):
        alphabet = 'abcdefghijklmnopqrstuvwxyz'
        private_key = 23
        decry_message = ""
        key = int(public_key / private_key)

        for c in msg:
            position = alphabet.find(c)
            new_position = (position - key) % len(alphabet)
            new_character = alphabet[new_position]
            decry_message += new_character
        return decry_message
```

7.3. Test the above functions. The outcome should be similar as below:

```
Please enter message: hello
Encrypted message:  axeeh
Decrypted message:  hello
```

8. Caesar decipher (task to be solved alone)

Write a Python function that got a Caesar encrypted message and gives back the key and the "human readable" message. Use the: ***iqfihhih*** encrypted message as an example input.

9. Factorization

There is no known method which can speed up factorization. This is the basis for reliability of public key cryptography. The next Python program tries to demonstrate the time requirement of this process.

9.1. Create a function which performs the factorization. It checks all numbers between 2 and n/2 (n is the parameter of the function). You can use the modulo (%) operator to check whether dividing *n* leaves remainder or not. (task to be solved alone)

9.2. Add a timer to benchmark how long your code takes to return the answer.

```
from time import time

#start the timer

start = time()


n = int(input('N: '))

factorization(n)          #this function has been implemented in 9.1.


#stop the timer

end = time()
```

```
print("Elapsed time: {} seconds".format(end - start))
```

9.3. Run your code and measure how long it takes to find the factors of 5, 6, and 7-digit numbers (on an average PC it takes some seconds).

References

[1] J.F. Kurose, K.W. Ross, Computer Networking, A Top-Down Approach, 6th ed., Pearson, 2013.

[2] McManus S, Cook, M. Raspberry Pi for Dummies, 3rd edition, Wiley, 2017.

[3] Raspberry Pi projects, https://projects.raspberrypi.org/en/projects/octapi-public-key-cryptography. Accessed on 05/09/2020.

[4] Raspberry Pi projects, https://projects.raspberrypi.org/en/projects/octapi-brute-force-enigma. Accessed on 05/09/2020.

This work was supported by the construction EFOP-3.4.3-16-2016-00021. The project was supported by the European Union, co-financed by the European Social Fund.