Lab 3: Symmetric Encryption Algorithms

1 Aim

The learning objective of this lab is for students to become familiar with the concepts of the symmetric encryption algorithm. After finishing the lab, students should be able to gain first-hand experience with symmetric encryption algorithms, operation modes, paddings, and Initial Vector (IV). Moreover, students will be able to use Python and write functions to realize security and performance analysis for a range of ciphers and operation modes.

2 Lab Duration

4 hours (two supervised hours in the lab, and two additional unsupervised hours).

3 Overview

There are three types of encryption algorithms:

1. **Symmetric ciphers:** all parties use the same key to encrypt and decrypt data. Symmetric ciphers are typically very fast and can process a very large amount of data.
2. **Asymmetric ciphers:** senders and receivers use different keys. Senders encrypt with public keys (non-secret) whereas receivers decrypt with private keys (secret). Asymmetric ciphers, such as RSA, are typically very slow and can process only very small payloads.
3. **Hybrid ciphers:** The two types of ciphers above can be combined in construction that inherits the benefits of both. An asymmetric cipher is used to protect a short-lived symmetric key, and a symmetric cipher (under that key) encrypts the actual message.

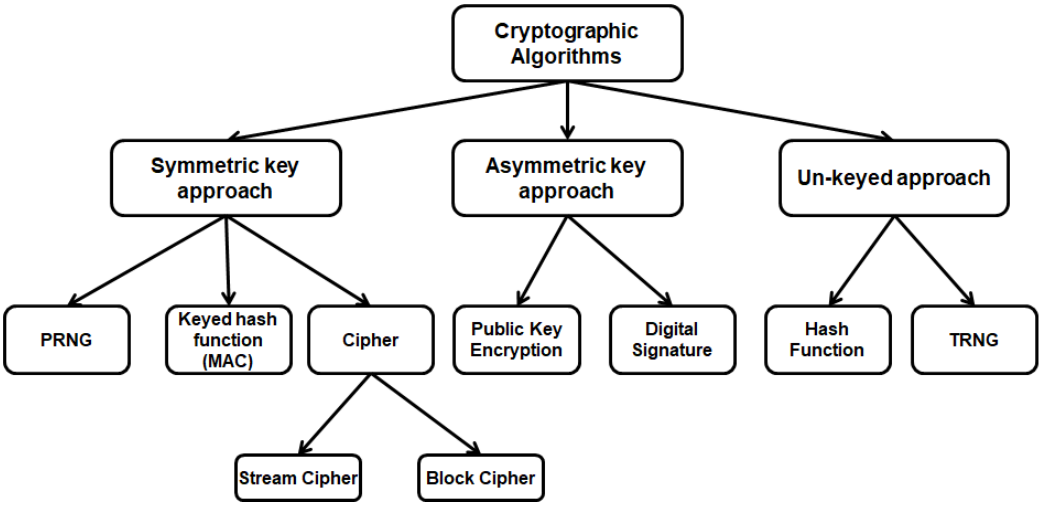

Figure 1: Classification of existing cryptographic algorithms

In addition, symmetric ciphers can be divided into two types of symmetric ciphers:

1. **Stream ciphers:** the most natural kind of ciphers that encrypt data by mixing it with a produced keystream. Examples of stream ciphers are ChaCha20 and XChaCha20 and Salsa20.
2. **Block ciphers:** ciphers that can operate on a fixed amount of data and blocks are processed according to a specific operation mode. The most important block cipher is AES, which has a block size of 128 bits (16 bytes).

This means that symmetric encryption algorithms can be performed at the block level or in stream mode. In stream cipher mode, the data is mixed (xor-ed) with a pseudo-random stream called "key-stream", usually at the byte level; for every byte of data, a random byte is generated and then, both bytes are xor-ed, and the output byte is transmitted as the cipher-byte. However, in the case of block cipher, the data is divided into blocks of fixed size, usually 128 bits. The stream-and-block-ciphering process is presented below.

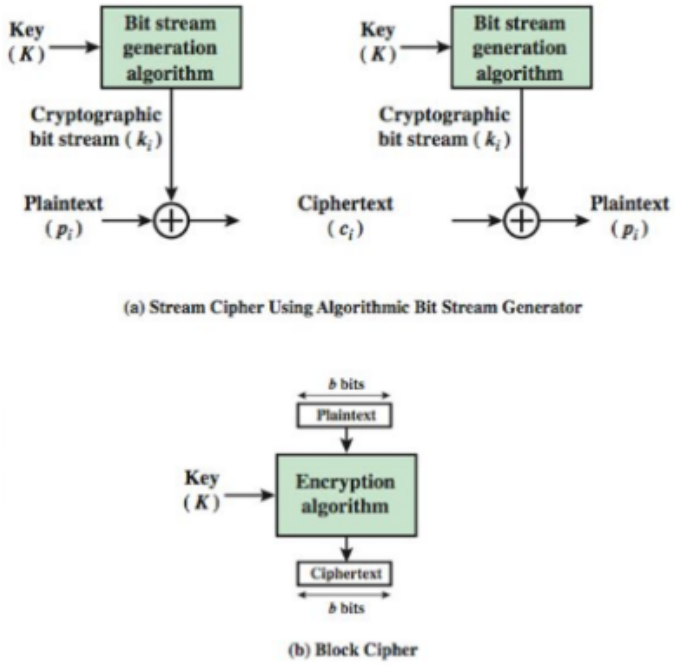

Figure 2: Stream Cipher versus Block Cipher

In general, a block cipher is mostly useful only together with a mode of operation, which allows one to encrypt a variable amount of data. Some modes (like OFB and CTR) effectively turn a block cipher into a stream cipher.

4 Objectives

At the end of this lab, you should understand:

- How to use a range of modern ciphers and operation modes.
- how to apply certain encodings, such as Base64, hex, and binary.
- How to write a Python script to perform security and performance analysis.

In addition, be sure to add titles, axis labels, and legends (where appropriate) to the plots.

5 Lab Environment

In this lab, we will use different packages of `Crypto` library such as `Crypto.Cipher`, which contains a set of modern symmetric cipher algorithms such as AES and they use for protecting data.

To install and validate the installation of this library with Colab:

```
!pip install pycryptodome
!pip install pycryptodomex
!pip install pycryptodome-test-vectors
!python3 -m Cryptodome.SelfTest
```

Note: list(x) can be used to convert bytes vector x to a list of integer values (works in Python 3.7). The following is a list of tasks that you should perform.

Exercise 1: Encoding with base64

Base64 is an encoding scheme that uses 65 printable characters (26 lowercase letters, 26 uppercase letters, 10 digits, characters + and '/', and a special character '='). Base64 allows the exchange of data with limited encoding problems.

To encode with base64, the following command is used:

```
from base64 import b64encode
y = b64encode(x)
```

To decode with base64, the following command is used:

```
from base64 import b64decode
x = b64decode(y)
```

1. Encode a text file containing an arbitrary password with base64 and store it in another file.
2. Decode this file (Base64) and store the output in another text file called "decoded.data".
3. How can an attacker find your password from this file? Is base64 a safe way to protect a password?

Exercise 2: Working with Random Number

To Generate random bytes in Python, we will use the method "get_random_bytes" of `Crypto.Random library`.

4. Generate 100 random bytes and show them on the screen (hexadecimal representation).
5. Generate 200 random bytes and store them (hexadecimal representation) in a file named "rand.txt".
6. Generate 300 random bytes and Encoding them into Base64 and store them in a file called "rand-Base64.txt".

Exercise 3: AES Primitives

1. Let AES-MC be a modified version of AES in which the ***MixColumns*** operation is eliminated. What is the problem with this scheme?
2. Let AES-SR be a modified version of AES in which the ***ShitRows*** operation is removed. What is the problem with this cipher scheme?
3. Let AES-SB be a modified version of AES in which the ***SubBytes*** operation is removed. What is the problem with this cipher scheme?
4. Let AES-RK be a modified version of AES in which the ***addition round key*** operation is eliminated. What is the problem with this scheme?

Exercise 4: Encryption using different block ciphers

In this task, you will test various encryption block cipher algorithms. To do this with the `crypto` library, you have to create (instantiate) a cipher object with the **new()** function that exists in the module cipher (`Crypto.Cipher`). The parameters of this method are as follows:

1. key (bytes)- The first parameter is the cryptographic key (a byte string).
2. mode– The second parameter is the selected operation mode.
3. iv- represents the initialization vector and is used for all operation modes, except ECB. It is as long as the block size (e.g. 16 bytes for AES). If not present, the library creates a random IV value.

Different block ciphers are defined at the module level. Starting by the cipher name and then the operation mode. For example, `Crypto.Cipher.AES.MODE_CBC`. Note that not all ciphers support all modes.

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
key = get_random_bytes(16)
cipher = AES.new(key, AES.MODE_CBC)
```

To encrypt/decrypt data, the method encrypt() (and likewise decrypt()) are used, respectively. Indeed, certain operation modes cipher object expect data to have a length multiple of the block size (e.g. 16 bytes for AES). You may need to use `Crypto.Util.Padding` to align the plaintext with the right boundary.

For example, to encrypt with AES mode CBC for a message, the following command is used:

```
data = b"secret"
ct_bytes = cipher.encrypt(pad(data, AES.block_size))
iv = b64encode(cipher.iv).decode('utf-8')
ct = b64encode(ct_bytes).decode('utf-8')
result = json.dumps({'iv':iv, 'ciphertext':ct})
print(result)
```

To decrypt a ciphertext with AES mode CBC, the following command is used:

```
try:
    b64 = json.loads(json_input)
    iv = b64decode(b64['iv'])
    ct = b64decode(b64['ciphertext'])
    cipher = AES.new(key, AES.MODE_CBC, iv)
    pt = unpad(cipher.decrypt(ct), AES.block_size)
    print("The message was: ", pt)
except (ValueError, KeyError):
    print("Incorrect decryption")
```

Please replace the AES `cipher` with another such as Blowfish, CAST-128, DES, triple-DES (TDES). In this lab, at least 3 different ciphers will be tested. These ciphers will be independent of the operating modes that will be tested in the next lab.

Exercise 5: Security and Performance Analysis Functions

1. **Distribution Analysis**: Write a function to create a histogram that shows how often each byte value (0-255) appears in an input byte vector, which represents ciphertext. What do you expect to see when you analyze the byte distribution of encrypted data? Explain your expectations.

2. Entropy Calculation: Develop a function to compute the entropy of a given byte vector. Calculate the entropy 1000 times using different random secret keys and messages of 256 bytes each. Do the results you obtain support or contradict your previous assumptions about byte distribution in encrypted files? Describe the findings.

3. **Bit Difference Calculation**: Write a function that calculates the percentage of differing bits between two input vectors of the same length. This function will be used to assess independence, message avalanche effect, and key avalanche effect that are requested in the following questions.

4. **Independence Validation**: Develop a script to verify if AES maintains independence between plaintext and ciphertext blocks. Use this script to analyze 1000 random block messages and secret keys. Present a plot displaying your measured results and summarize your observations briefly.

5. **Key Avalanche Effect**: Write a script to determine if AES exhibits the key avalanche effect, meaning that even a small change in the secret key should significantly affect the ciphertext. Test this effect with 1000 random block messages and secret keys, and provide a plot illustrating your findings. Summarize your observations concisely.

6. **Message Avalanche Effect**: Write a script to evaluate if AES achieves the message avalanche effect, where modifying any bit in the input plaintext block should lead to a significant difference in the ciphertext. Perform this analysis with 1000 random block messages and secret keys, and present a plot showing your results. Summarize your observations briefly.

Exercise 6: Stream Cipher vs. Block cipher

In general, stream-cipher algorithms require a nonce, which must not be reused across encryptions performed with the same key. In fact, a block cipher can be used as a stream cipher when the block cipher is employed to produce a key-stream sequence, which is the case of Output FeedBack (OFB) and Counter

(CTR) operation modes. The security of a stream cipher is based on different metrics that depend on the quality of the produced key-stream sequences, which should exhibit high non-linearity, long periodicity, high level of randomness, and high uniformity degree.

An example of a stream cipher such as ChaCha20, which was designed by Daniel J. Bernstein. The secret key is 256 bits long (32 bytes). Besides, there are three variants, defined by the length of the nonce, and they are listed in the following:

Nonce length	Description	Max data	If random nonce and same key
8 bytes (default)	The original ChaCha20 designed by Bernstein.	No limitations	Max 200 000 messages
12 bytes	The TLS ChaCha20 as defined in RFC7539.	256 GB	Max 13 billions messages
24 bytes	XChaCha20, still in draft stage.	256 GB	No limitations

This is an example of how ChaCha20 (Bernstein's version) can encrypt data:

```
import json
from base64 import b64encode
from Crypto.Cipher import ChaCha20
from Crypto.Random import get_random_bytes
>>>
plaintext = b'Attack at dawn'
key = get_random_bytes(32)
cipher = ChaCha20.new(key=key)
ciphertext = cipher.encrypt(plaintext)
>>>
nonce = b64encode(cipher.nonce).decode('utf-8')
ct = b64encode(ciphertext).decode('utf-8')
result = json.dumps({'nonce':nonce, 'ciphertext':ct})
print(result)
{"nonce": "IZScZh28fDo=", "ciphertext": "ZatgU1f30WDHriaN8ts="}
```

And this is how you can decrypt it:

```
import json
from base64 import b64decode
from Crypto.Cipher import ChaCha20
# We assume that the key was somehow securely shared
try:
    b64 = json.loads(json_input)
    nonce = b64decode(b64['nonce'])
    ciphertext = b64decode(b64['ciphertext'])
    cipher = ChaCha20.new(key=key, nonce=nonce)
    plaintext = cipher.decrypt(ciphertext)
    print("The message was " + plaintext)
except (ValueError, KeyError):
    print("Incorrect decryption")
```

7. Encrypt a message using the ChaCha20 stream cipher. Additionally, encrypt the same message using the AES block cipher with the CBC (Cipher Block Chaining) operation mode.

8. **Stream Cipher Evaluation**: Develop a script to assess whether a stream cipher algorithm like ChaCha20 can maintain independence, message avalanche effect, and key avalanche effect for 1000 randomly generated keys.

9. **Initial Vector Sensitivity**: Write a script to investigate whether a stream cipher can preserve initial vector (IV) sensitivity. In this test, consider that the original IV is altered in one bit.

10. **Corrupted ciphertext**: Unfortunately, a single bit of any byte in the encrypted file has been corrupted. You can intentionally introduce this corruption. Decrypt the corrupted file (encrypted) using the correct key and IV. Before conducting this task, predict how much information you can recover by

decrypting the corrupted file if the encryption mode is AES or ChaCha20, respectively. After completing the task, verify if your prediction was accurate and discuss the implications of any differences.

6 Performance Test

11. **Block Cipher Execution Time vs. Message Length**: Create a script to assess the performance of three implemented ciphers from the Crypto.Cipher library, including AES, single DES, and TDES, under a specific operation mode. Measure the execution time (or throughput, calculated as Message length divided by Execution Time) against varying message lengths. Provide a plot illustrating the measured results for all ciphers. Summarize your observations briefly.

12. **Block cipher: Encryption and Decryption Performance**: Write a script to evaluate the performance of encryption and decryption using a specific cipher, possibly AES. In this test, measure the execution time (or throughput) of both encryption and decryption modes as a function of message length. Present a plot displaying the measured results and summarize your findings concisely.

13. Performance of stream cipher (ChaCha20 or Salsa20): Develop a script to gauge the performance of encryption using ChaCha20 or Salsa20. Calculate the execution time (or throughput) of ChaCha20 or Salsa20 encryption as a function of message length. Generate a plot to illustrate the measured results, and briefly summarize your observations.

14. **Block Cipher vs. Stream Cipher Performacence**: Write a script to assess the performance of encryption using either a block cipher (e.g., AES) or a stream cipher (e.g., ChaCha20). Measure the execution time (or throughput) of encryption with both block and stream ciphers relative to message length. Create a plot to display the measured results and provide a brief summary of your observations.

Good Luck