Politecnico di Torino

Facoltà di Ingegneria

Laboratory activities on crypto cores

Operating systems

Authors: group 12

Luigi Palmisano (319867)
Alberto Castronovo (290238)
Giuseppe Cutrera (318954)
Matteo Carnevale (317718)

January 29, 2024

Contents

CHAPTER 1

Self-evaluation questions

1.1 Questions

1. Which encryption did AES superseed when it was established by the U.S?
   (a) DES
   (b) SHA
   (c) Rijndael
   (d) MD5
2. What is the safest key size used in AES against brute-force attacks?
   (a) 192
   (b) 128
   (c) 256
   (d) 512
3. Is AES safe against every possible attack?
   (a) Yes, because it's the safest and has currently no weaknesses
   (b) Partly, because it's safe from frontal attacks but it's weak against side-channel attacks
   (c) Partly, because it's safe from all side-channel attacks but it's weak from frontal attacks
   (d) No, because it's one of the easiest to crack
4. Which AES is/are considered quantum resistant?
   (a) 192
   (b) 128 and 192
   (c) 256 and 192
   (d) 256
5. Why is it beneficial to employ a large block size?
   (a) It allows to use a longer key, and therefore make the protocol more secure
   (b) Block size is irrelevant with modern algorithms

(c) It makes the protocol less susceptible to birthday attacks

(d) To increase encryption speed, as less blocks will be needed for the same data

6. (multiple answers) Why is ECB mode not recommended for applications where security is critical, like bank accounts information?

(a) It can leave plaintext data patterns in the ciphertext

(b) It employs a smaller block size

(c) It is very slow and more susceptible to timing attacks, compared to other modes

(d) It is very malleable, because each block is encrypted independently

1.2 Solutions

1. (a)
2. (c)
3. (b)
4. (d)
5. (c)
6. (a and d)

CHAPTER 2

Crypto core device and interface

This section aims at briefly presenting the structure and internal operations of the emulated crypto core device and its corresponding driver, in order to ease the execution of the laboratory exercises presented in the following sections.

2.1 The device

The device is implemented in the *crypto_core.c* and *crypto_core.h* files, located alongside the other device files in QEMU as shown in the installation guide. The interface is composed of several $registers$, each one being located in a fixed and specific location in main memory:

1. *id*: a read-only register that stores the device ID. It is hard-coded in the *.c* file. This is primarily used as a quick way to check if the device is working when QEMU is executed;
2. *mode*: must be set to 0 to perform *encrypt* operations, and any other positive value (in the $uint32\_t$ range) for *decrypt* operations;
3. $format$: set to 0 to employ ECB encryption/decryption mode, 1 for CBC mode and any other positive value in the $uint32\_t$ range for CTR mode;
4. *start*: set to 1 to start the configured operations. The other registers must be set beforehand. The value of this register is reset to 0 when the operation is completed;
5. *valid*: set to 1 by the device when the operation is finished. Must be reset to 0 by the user, even though it will not compromise the success of future operations otherwise (it will just fail at signalling when the operation will be completed);
6. *key*: write-only register that stores the secret key;
7. *iv*: write-only register that stores the secret initialization vector;
8. *in*: write-only register that stores the input plaintext or ciphertext to be encrypted or decrypted. The operation must be performed one block at a time.
9. *out*: read-only register that stores the operation result.

The same file contains all the functions required for the $AES256$ operations. Other modes could be employed, by changing the preprocessor directives accordingly.

The device is located in main memory at the base address $0x8000000$, with a size of $0x200$ bytes, which is a little higher than required (the last word is located at an offset of $0x160$), to allow room

for future development. As each word in memory has a size of 4 bytes (32 bits), there are multiple words defined for registers *key*, *iv*, *in* and *out* (respectively 8, 4, 4, 4). The device also includes some more registers, ending with *_char* and *_hex*, to provide multiple ways to interface the same registers, as explained in the following section.

2.2 The driver

The driver provides the user with an easy way to interface with the crypto core, via device files. The device files can be accessed from the directory $/sys/bus/platform/devices/crypto_core@8000000$. In order to make the interface as clear as possible, several device files, instead of a single one, are provided. One file for each of the previously defined words is present, which means that multiple files are defined for *key*, *iv*, *in* and *out*. Moreover, to make interfacing these four longer registers easier, *_char* and *_hex* device files are provided for each one.

1. *_char*: allows the user to write a 16-characters long (or 32-characters long, for the secret key) buffer into the file, which will be written by the driver into the 4 (or 8) separate variables. The read operation, instead, returns a space-separated string of unsigned integers, because some encrypted characters could not be printable and would result in a messy representation if printed on screen;

2. *_hex*: allows the user to read or write a 32- (or 64-) characters long buffer of hexadecimal characters **(uppercase)**, with each pair representing a byte.

The indexing of *_char* and *_hex* files is big endian and with crescent index of register. For example, if the *in_hex* register was given the string $A05B235699AB124C4481BBAF2109113A$, the most significant byte of *in_0* would be $A0$ and the least significant byte of *in_3* would be $3A$.

CHAPTER 3

Emulation of a criptocore using QEMU

3.1 Exercise 1: encrypt and decipher a message

In this exercise you are going to play around with the cryptocore functionalities using the QEMU environment. Throughout the laboratory we will work with the plaintext message:

```
ImFlyingOverSeas
```

You are asked to:

1. Setup the cryptocore with QEMU: start the environment using the script *start_qemu.sh*, load the device driver into the kernel and launch the test program, or create your own.
2. Once the device has been configured, encrypt the message above using the following key (hex representation):

   ```
   6a2580412d38023056983eb6ce874e51d64baead4cd485d634d51ad47e07f56e
   ```

   Try ECB mode first.
3. Now let's move to CTR and check the result. Use the following IV (hex representation):

   ```
   ec2de7efc24283a261840101c6fe5295
   ```

   Use the same key as before.
4. Finally, try using CBC mode with the same IV and key.

Solutions:

1. CB B0 59 C0 16 9A C5 26 9A 79 12 86 57 AD 19 90
2. 6F EB E7 E8 FD B2 1A F4 18 83 4F 03 86 FD 96 C0
3. 6E E7 8E E7 B2 93 0F 9C D3 16 96 66 FB 08 F0 E1

CHAPTER 4

Bit-flipping attack on CTR mode

4.1 CTR encryption and decryption

The CTR mode of operation follows the structure shown in figure 4.1.

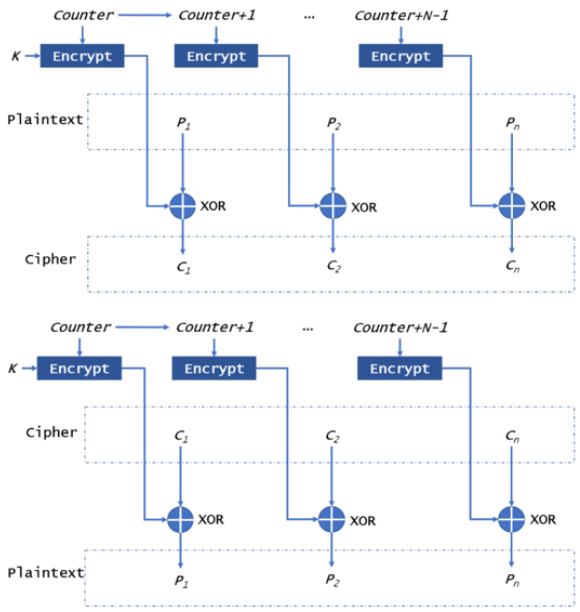

Figure 4.1: AES-256 CTR mode structure, courtesy of www.highgo.ca.

For encryption, each block of the plaintext is XOR'ed with an *Encrypt* block of the same size, obtained by performing operations that involve the secret key and the value of a counter. The counter has the same number of bits as the block size, and its value is incremented for each consecutive block. All blocks employ the same secret key, which should be changed at least after every $2^{n/2}$ encryption blocks to ensure proper protection against birthday attacks, with n being the block size. For decryption, the operation is the same, with the ciphertext being XOR'ed with the *Encrypt* blocks to re-obtain the plaintext at the output. This is done by exploiting the fact that, with XOR operations, $B \oplus (A \oplus B) = A$, as shown in table 4.1.

This mode has several benefits, including support for parallel computing, ease of implementation (encryption and decryption perform the same operation) and robustness, as any errors in a specific block (*bad blocks*) would only affect that specific blocks, leaving the following ones unaffected.

P_1	E	$C = P_1 \oplus E$	$P_2 = C \oplus E = P_1$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Table 4.1: Summary of the XOR operations employed in CTR encryption and decryption.

4.2 Premises for the attack

The goal of the bit-flipping attack is to **make the decrypted plaintext be a message of your choice, regardless of the secret key or counter starting value (IV).** In other words, this attack, under the following premises, allows us to write an arbitrary text to the output register of the crypto core when decryption is performed. In order to perform the attack, we must be able to perform the following operations:

1. read the encrypted version of a known plaintext;
2. be able to alter the content of the ciphertext before the decryption.

For example, say that we are logging into a website. The website will internally exchange some information about our profile, including possible flags like $admin = false$. These informations are encrypted via SSL, and the AES algorithm is a possible choice for the protocol. If we, the attackers, are able to access the ciphertext corresponding to the plaintext $admin = false$ and alter it, we can possibly make the website interpret it as $admin = true$, and therefore be able to gain unintended permissions.

4.3 Exploiting XOR operations

Suppose that we have two plaintexts P_1 and P_2. P_1 is the initial known plaintext that the crypto core will encrypt, and P_2 is the plaintext that we want the crypto core to decrypt and return at its output. If we send P_1 to the crypto core, we can collect the ciphertext $C_1 = P_1 \oplus E$. Before altering the content fo C_1, we evaluate the XOR operation between the two known plaintexts, $X = P_1 \oplus P_2$. We use this information to tamper the ciphertext, and inject a new ciphertext $C_2 = C_1 \oplus X$. The crypto core will perform decryption on the tampered ciphertext C_2, returning a plaintext P_3, which will correspond to P_2, as shown in table 4.2.

P_1	P_2	E	$X = P_1 \oplus P_2$	$C_1 = P_1 \oplus E$	$C_2 = C_1 \oplus X$	$P_3 = C_2 \oplus E$
0	**0**	0	0	0	0	**0**
0	**0**	1	0	1	1	**0**
0	**1**	0	1	0	1	**1**
0	**1**	1	1	1	0	**1**
1	**0**	0	1	1	0	**0**
1	**0**	1	1	0	1	**0**
1	**1**	0	0	1	1	**1**
1	**1**	1	0	0	0	**1**

Table 4.2: Summary of the XOR operations used to alter the decrypted plaintext.

4.4 Exercise requirements

Write a C program that performs a bit-flipping attack on the crypto core in CTR mode. You can use the functions provided in *crypto_functions.h* to interact with the virtual device, or you can write your own functions.

1. The program must be able to perform the attack **without knowing the secret key or IV.** If you implement encryption, attack and decryption in the same file, then make sure that **the attack does not use their value in any way.**
2. The initial plaintext $P1$ must be arbitrary (the program should be able to function regardless of its specific content), but it is known (or readable) to the attacker.
3. If you want to implement encryption and decryption operations in the same file, **remember to reset the IV value before decryption!** Otherwise you will get wrong results, because the encryption operation changes the value of the IV registers according to the internal counter.

CHAPTER 5

Vulnerabilities of the ECB mode

Exercise 3: Use python to highlight the ECB vulnerabilities

Step 1: Create a New Python File

Create a new Python file (e.g., encryption_demo.py) and add the following content:

Step 2: Import Required Modules

Import the necessary modules for your script.

```
import subprocess
import os
import secrets
```

Step 3: Define Function to Generate Random Keys and IVs

```
def generate_random_bytes(length):
    return secrets.token_hex(length)
```

Step 4: Define Function to Call the C Program

```
def call_c_program(program_name, *args):
    str_args = [str(arg) for arg in args]  # Convert all arguments to strings
    subprocess.run(["./" + program_name] + str_args, capture_output=True, text=True)
```

Step 5: Define the Main Function to Demonstrate Encryption Modes

```
def demonstrate_encryption_modes():
    key = generate_random_bytes(16)  # 32-character key (256 bits)
    iv = generate_random_bytes(8)    # 16-character IV (128 bits)
    plaintext = '1234567890ABCDEF'   # 16-character plaintext

    print(f"Using Key: {key}")
    print(f"Using IV: {iv}")
    print(f"Plaintext: {plaintext}")

    # Encrypt with ECB mode
    call_c_program("perform_all_operations", key, iv, plaintext, 'ECB', '0')

    # Encrypt with another mode (e.g., CBC)
    call_c_program("perform_all_operations", key, iv, plaintext, 'CBC', '0')
```

Step 6: Add the Main Block

```
if __name__ == "__main__":
    demonstrate_encryption_modes()
```

Step 7: import on QEMU and run it

Now you have to pass the python file to QEMU in the same wave done in the previous exercise and run the program.

`Tips`: Remember to run insmode crypto-core.ko before executing the python program.