CS 445: Applied Cryptography and Computer Systems Security

Block Cipher Lab

Team Members

Chinwe Ibegbu
Nana Kofi Boakye

October 2022

Report

What We Did and Observed

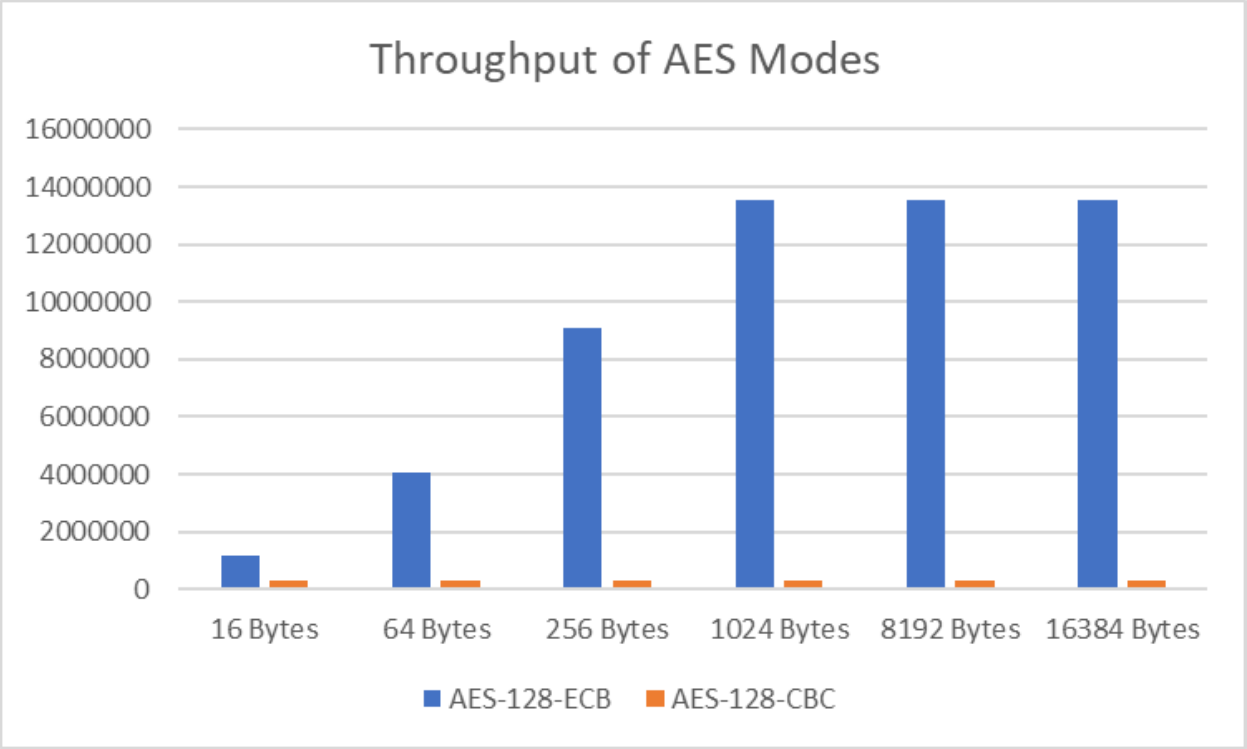
Task 1: For Task 1, Chinwe and I spent quite some time reading the question and looking at what exactly the Task wanted us to do. It helped us to highlight and break down the question. With that, we then looked at Cryptographic libraries that would be of help to us. The task made mention of the Cryptographic library and so we employed that one and installed it into our local environment. Using the OS library we created a random key and Initialization Vector (IV) as global variables throughout our file. We then made two separate functions for ECB and CBC and we made the functions such that it took system arguments for the sake of defining the plaintext file. The file would then open the file, read it as a binary file and save it to a byte array. Each line in the file (in binary) is then saved in the byte array. We then calculated the bit length and the number of blocks. That determined whether or not the plain text file needed padding. If padding was needed, we used PKCS7 from the Cryptography library. Next would be the encryption and that was dependent on the function. For ECB or CBC we partitioned the lines into blocks with a size of 128 and then using the Cryptography library created a Cipher which then encrypted each block. Each block was then compiled with the next. The difference between the two Encryption modes was on the Cipher. For ECB, we passed a mode suitable for ECB encryption and likewise for CBC. For both encryptions, a for loop was used and in the case of CBC, after the first encrypted block, we passed the previously encrypted data to the subsequent block. Once the encryption was done, we wrote the compiled encrypted data into a file.

Task 2: Task 2 was relatively straightforward. For submit(), the program requests user input and makes it an array. It then loops through the user input for any ';' and '='. It then encodes them using the "urllib" library. We then put the characters and letters back together. After we have a suitably encoded string, we prepend and append the encoded string with suitable data which is then converted to bytes for padding and encryption. The encryption was done with the AES128 algorithm in a CBC mode with a random IV. We then print the cypher text.

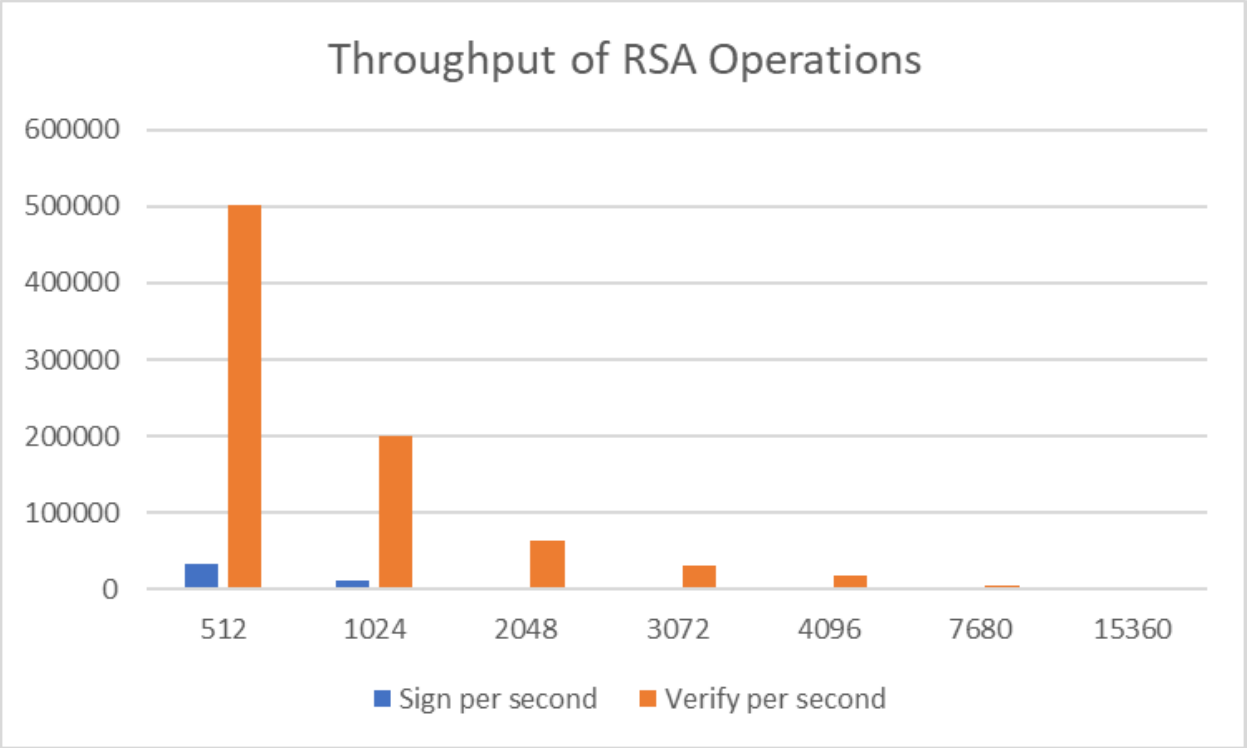
For verify(), the function took a cypher text, decrypted it, and then converted it from bytes back to characters in the UTF-8 standard. We then searched for the phrase ";admin=true;". Depending on whether or not that phrase was present, a true or false was returned. The outcome is then printed out.

Task 3: For task 3 we thought that it would be a Python script. However, it was realised that this was more of a terminal-oriented solution. Thus after reading the documentation for Openssl, we got the commands that would give us the results we wanted.

For AES128, we passed "***openssl speed -evp aes-128-ecb***" and "***openssl speed -evp aes-128-cbc***" for the throughput for ECB and CBC respectively. The returned data was then placed in Excel and then using a Bar Chart, we then produced this graph



For the RSA operations, we ran the command "***openssl speed rsa***". The data was then compiled and using a bar chart once again, this was the outcome:



Questions

1. For task 1, viewing the resulting ciphertexts,
    a. What do you observe?
    b. Are you able to derive any useful information about either of the encrypted images?
    c. What are the causes for what you observe?

2. For task 2,
    a. Why is this attack possible?
    b. What would this scheme need in order to prevent such attacks?

3. For task 3,
    a. How do the results compare?
    NOTE: Make sure to include the plots in your report.

Answers

Task 1

1. A bunch of data in weird symbols when we tried to open the file

2. Not really. We weren't able to derive anything of use
3. I believe the cause was the encryption of the file and then the cypher text being in Binary format

Task 2

1. The reason that this task is vulnerable to that specific attack is that it is using AES in CBC mode. Thus, flipping even a single bit in one of the cypher texts can completely scramble the original plaintext. This is because, in CBC mode, the cypher text of one block was dependent on the plaintext of the current block and the cypher text of the previous block. In summary, there is no way for the sender or receiver to ensure integrity and authentication.
2. This scheme would need a way to ensure integrity and authentication to combat this issue. A Message Authentication Code (MAC) can be used for such purposes.

Task 3

1. For the AES throughput comparison, ECB was giving higher levels of throughput as compared to CBC when running against a 128 AES algorithm.
2. As for RSA, it was quite obvious that for both operations, as the bit key size increased each operation took less time. However, regardless of the key size, Verify per second had a higher throughput than that of Sign per second

Code Appendix

```
from pydoc import plain
import sys, os
import urllib.parse as url_encode
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
   modes

# Create key and initialisation vector of size 128 bits
key = os.urandom(16)
iv = os.urandom(16)

# AES Decryptor and Encryptor
AES_algorithm = algorithms.AES128(key)
cipher = Cipher(AES_algorithm, mode=modes.CBC(iv))
encryptor = cipher.encryptor()
decryptor = cipher.decryptor()

def encrypt_ecb():
    plaintext_file_name = sys.argv[1]
    plaintext_file = open(plaintext_file_name, mode='rb')
    plaintext_data = bytearray()
    for line in plaintext_file:
        plaintext_data += line

    # Display plaintext and number of blocks
    # print(lines)
    bit_length = plaintext_data[0].bit_length()
    total_bit_length = len(plaintext_data)*8
    num_blocks = total_bit_length/128

    print("\nBit length:", str(bit_length))
    print("Total bit length:", str(total_bit_length))
    print("Number of blocks:", str(num_blocks), "\n")

    plaintext_file.close()

    # Pad plaintext if necessary
    if(total_bit_length%128 != 0):
```

```
        padder = padding.PKCS7(128).padder()
        padded_lines = padder.update(plaintext_data) +
    padder.finalize()

        # Print unpadded and padded versions
        new_total_bit_length = len(padded_lines)*8
        new_num_blocks = new_total_bit_length/128

        print("New total bit length:", str(new_total_bit_length))
        print("New number of blocks:", str(new_num_blocks), "\n")

        # Update the lines variable
        lines = padded_lines
        total_bit_length = new_total_bit_length
        num_blocks = new_num_blocks

    else:
        print("Sufficient bits originally available")


    # Create array to contain cipher text in binary
    ciphertext_data = bytearray()

    # Encrypt the file
    for i in range(0, int(num_blocks)):
        first_byte_index = i*16
        last_byte_index = i*16 + 16

        current_block = lines[first_byte_index:last_byte_index]
        AES_algorithm = algorithms.AES128(key)
        cipher = Cipher(AES_algorithm, mode=modes.ECB())
        encryptor = cipher.encryptor()
        cipher_text = encryptor.update(current_block)
        ciphertext_data += cipher_text

    ciphertext_file = open('ecb_encrypted', 'xb')
    ciphertext_file.write(ciphertext_data)
    ciphertext_file.close()

    return 0
```

```
def encrypt_cbc():

    # Read the content of the plaintext file
    plaintext_file_name = sys.argv[1]
    plaintext_file = open(plaintext_file_name, mode='rb')
    plaintext_data = bytearray()
    for line in plaintext_file:
        plaintext_data += line

    # Display plaintext and number of blocks
    # print(lines)
    bit_length = plaintext_data[0].bit_length()
    total_bit_length = len(plaintext_data)*8
    num_blocks = total_bit_length/128

    print("\nBit length:", str(bit_length))
    print("Total bit length:", str(total_bit_length))
    print("Number of blocks:", str(num_blocks), "\n")

    plaintext_file.close()

    # Pad plaintext if necessary
    if(total_bit_length%128 != 0):
        padder = padding.PKCS7(128).padder()
        padded_lines = padder.update(plaintext_data) +
   padder.finalize()

        # Print unpadded and padded versions
        new_total_bit_length = len(padded_lines)*8
        new_num_blocks = new_total_bit_length/128

        print("New total bit length:", str(new_total_bit_length))
        print("New number of blocks:", str(new_num_blocks), "\n")

        # Update the lines variable
        lines = padded_lines
        total_bit_length = new_total_bit_length
        num_blocks = new_num_blocks
```

```
    else:
        print("Sufficient bits originally available")


    # Create array to contain cipher text in binary
    ciphertext_data = bytearray()

    # Encrypt the file
    for i in range(0, int(num_blocks)):
        first_byte_index = i*16
        last_byte_index = i*16 + 16

        current_block = lines[first_byte_index:last_byte_index]
        AES_algorithm = algorithms.AES128(key)
        if i > 0:
            iv = cipher_text
        cipher = Cipher(AES_algorithm, mode=modes.CBC(iv))
        encryptor = cipher.encryptor()
        cipher_text = encryptor.update(current_block)
        ciphertext_data += cipher_text

    # Create output file
    ciphertext_data = ciphertext_data.decode("utf-8")
    ciphertext_file = open('cbc_encrypted', 'xb')
    ciphertext_file.write(ciphertext_data)
    ciphertext_file.close()


    decryptor = cipher.decryptor()
    decrypted_data = decryptor.update(ciphertext_data)

    # plaintext_file = open('plaintext', 'xb')
    # plaintext_file.write(decrypted_data)
    # plaintext_file.close()
    return 1

# encrypt_ecb()
```

```
def submit():

    # Getting the user input and making it an array
    userInput = input("Please give me a string: \n")
    letters = [*userInput]

    # Encoding each ';' and '=' in the userInput
    for i in range(len(letters)):
        if letters[i] == ';' or letters[i] == '=':
            urlEncoded = url_encode.quote(letters[i])
            letters[i] = urlEncoded

    # Putting the characters back together into one variable
    encodedUserInput = ""
    for i in letters:
        encodedUserInput += i

    # Adding the appropriate prepend and appent to the encoded
   userInput
    prepend = "userid=456;userdata="
    append = ";session-id=31337"
    full_text = prepend + encodedUserInput  + append

    # Printing the output of the full_text
    print("Full configured text: ", full_text, "\n")

    # Converting full_text to bytes
    textInBytes = str.encode(full_text)

    # Padding the full_text
    padder = padding.PKCS7(128).padder()
    paddedText = padder.update(textInBytes) + padder.finalize()

    # Printing the padded text
    print("Padded text: ", paddedText, "\n")

    # Encrypting padded text to AES-128-CBC
    cipher_text = encryptor.update(paddedText) + encryptor.finalize()

    # Printing the cipher_text in bytes...
```

```
    print("AES Encrypted cipher text is: \n", cipher_text, "\n" )

    return(cipher_text)

def verify(cipher_text):
    plain_text = decryptor.update(cipher_text) + decryptor.finalize()
    plain_text = plain_text.decode("utf-8")
    print("The plain text is: \n", plain_text)
    index = plain_text.find(";admin=true;")
    if index > -1:
        return True
    else:
        return False

ct = submit()
result = verify(ct)
print("\nOutcome:", result)
```