Block Cipher

Kyle Jennings, Zarek Lazowski, Evan Morris

Questions

1. For task 1, viewing the resulting ciphertexts, what do you observe? Are you able to derive any useful information about either of the encrypted images? What are the causes for what you observe?
    a. The ECB encrypted files are still recognizable to what they were before they were encrypted. The only difference is that the colors are off. This is because the encryption only operates on 16 bytes at a time so the spatial relations between all the sections of data is preserved.
    b. For CBC there is no visual information preserved. This is because each block changes the encryption on the next block, effectively eliminating the limitation of a 16-byte encryption size.
2. For task 2, why is this attack possible? What would this scheme need in order to prevent such attacks?
    a. This attack is possible because the attacker has access to both the input and the encrypted form of the data. This means they can edit it while it's in the air. To prevent this, we could ensure that the entire message makes sense. During the edit, it destroys the rest of the message, so making sure the input is well-formatted could prevent this.
3. For task 3, how do the results compare? Make sure to include the plots in your report.
    a. For AES-CBC, the block size has a minor positive effect on throughput while the key size has a major negative effect.
    b. For RSA the key size greatly effects verify speed, with smaller keys translating to higher speeds. This relationship is also true for sign speed but is much less pronounced.

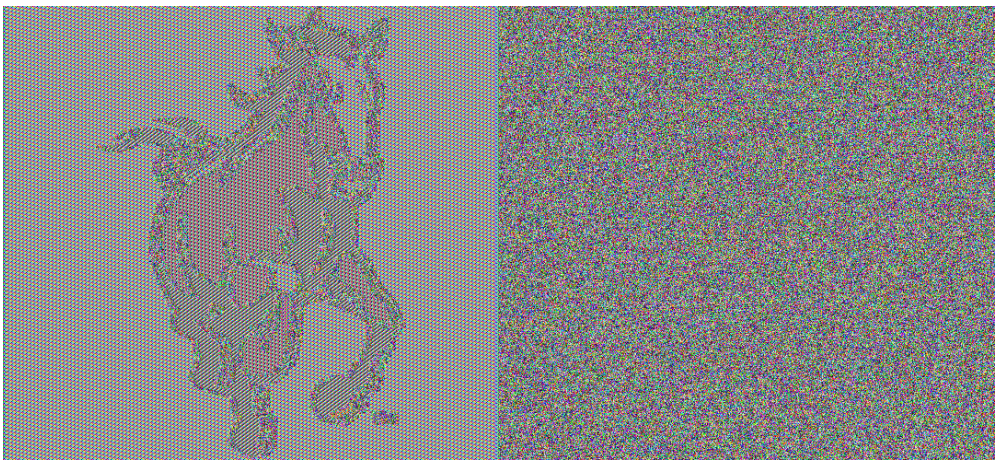
Task 1

The basic goal of the first task was to implement two different AES chaining algorithms to enable the encryption to work for more than one block at a time. This code can be seen in *task1.py* and *block_cipher.py*.

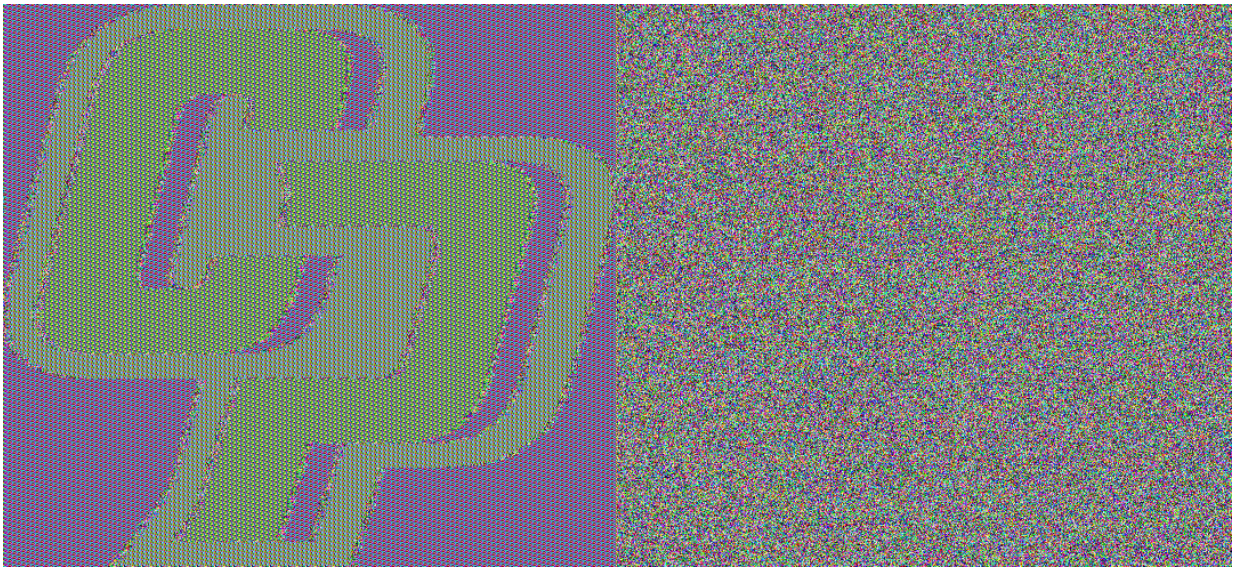
The first function we needed to implement was PKCS7 padding which makes sure that arbitrary sized data is 16-byte aligned. It does this by adding bytes to the end of the input until it is the correct size. More information about the algorithm can be found here (https://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS#5_and_PKCS#7).

After implementing padding, we can start using the encryption chaining. To encrypt anything, we use the python *cryptography* library. By default, this needs to have a chaining algorithm to work so we gave it a wrapper function that forces it to only encrypt 16 bytes at a time. Then we can implement ECB and CBC. The algorithms can be seen in their respective functions inside of *block_cipher.py* and also here (https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Electronic_codebook_(ECB), https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_block_chaining_(CBC)) . ECB is simple, just use the given key and encrypt each block. CBC is a bit more complicated; you must provide an initialization vector (IV) as well as the key and XOR the plaintext with that before encrypting for the first block. After the first block you use the previous encrypted block as the IV.

AES-ECB: AES-CBC:



AES-ECB: AES-CBC:



Task 2

In the second task we need to take advantage of the CBC algorithm to insert information into a message that we presumably intercepted in the air.

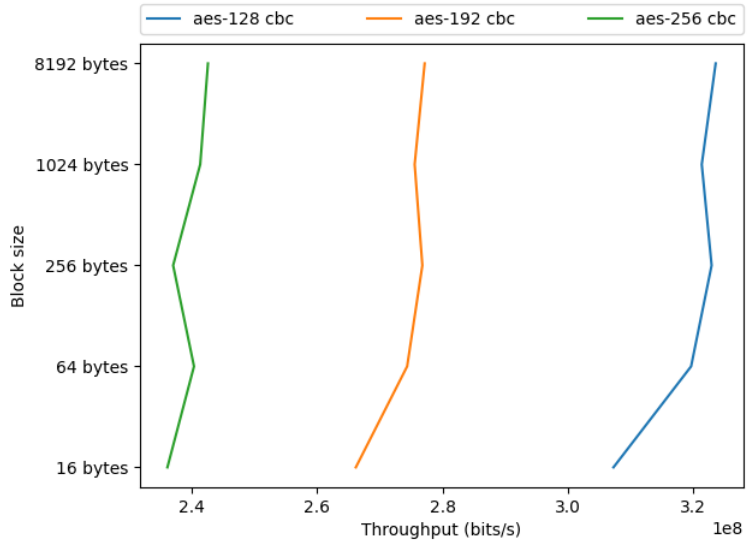
After writing the submit and verify functions (found in *task2.py*) we can start trying to break them. The verify function explicitly only check if ";admin=true" is the in decrypted message. This means we don't have to worry about anything other than getting that into the message.

To accomplish this, we take advantage of CBC decryption. Since CBC decryption uses the cipher text of the next block to decode the previous block, we can change that ciphertext to whatever we want. In this case we make it ";admin=true" and pad it to be 16 bytes. We can also control the input. So, in the input we make sure to have one of the encrypted blocks be all zero. That way when it decodes the ciphertext it ends up XORing the decrypted zero block with whatever we provided earlier. Since anything XORed with zero is itself, we end up inserting that edit we made to the ciphertext into the plaintext.

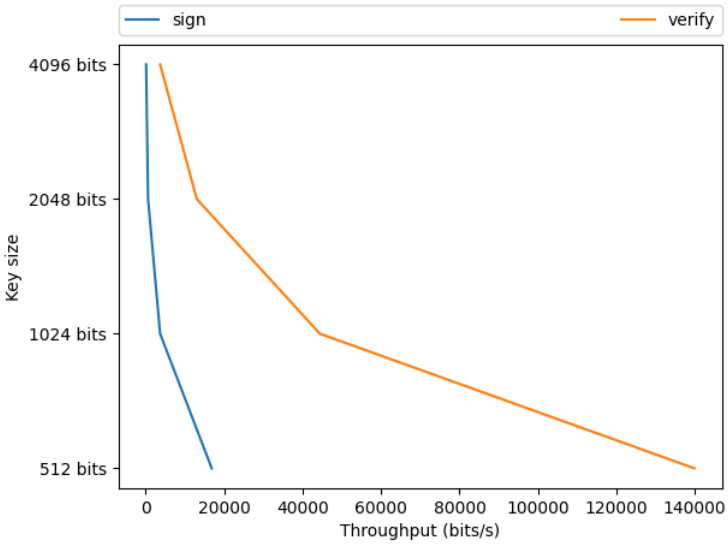
Task 3

The last task requires the running of *openssl rsa speed* and *openssl aes speed* and the graphing of the results. The program to generate the graph is simple and is included under *task1.py*.

AES:



RSA:



Code

block_cipher.py

```
from cryptography.hazmat.primitives.ciphers import (
    Cipher, algorithms, modes
)

BLOCKSIZE = 16


def pkcs7_pad(input: bytes, blocksize: int) -> bytes:
    """Pad a byte array according to the PKCS#7 standard.

    Args:
        input (bytes): array to pad
        blocksize (int): size of blocks in bits

    Raises:
        ValueError: blocksize is greater than 256

    Returns:
        bytes: the padded array
    """
    # check the size
    if blocksize > 2040:
        raise ValueError(f"Cannot pkcs7 pad, the blocksize is too large:
{blocksize}")
    if blocksize % 8:
        raise ValueError(f"Cannot pkcs7 pad, the blocksize is not divisible
by 8: {blocksize % 8}")

    # change to bytes instead of bits
    blocksize //= 8

    # get the amount needed to get to the next border
    remainder = BLOCKSIZE - (len(input) % blocksize)

    if remainder == 0:
        return input + bytes([blocksize for _ in range(blocksize)])
    return input + bytes([remainder for _ in range(remainder)])


def pkcs7_strip(input: bytes) -> bytes:
    """Strip trailing bytes according to the PKCS#7 standard.
```

```
    Args:
        input (bytes): padded input to be stripped

    Returns:
        bytes: the stripped array of bytes
    """
    return input[:-1 * int(input[-1])]


def aes128_encrypt(data: bytes, key: bytes) -> bytes:
    if len(data) != BLOCKSIZE and len(key) != BLOCKSIZE:
        raise ValueError("aes128 only deals with 128 bits at a time")

    encryptor = Cipher(algorithms.AES(key), modes.ECB()).encryptor()
    return encryptor.update(data) + encryptor.finalize()


def aes128_decrypt(data: bytes, key: bytes) -> bytes:
    if len(data) != BLOCKSIZE and len(key) != BLOCKSIZE:
        raise ValueError("aes128 only deals with 128 bits at a time")

    decryptor = Cipher(algorithms.AES(key), modes.ECB()).decryptor()
    return decryptor.update(data) + decryptor.finalize()


def ecb_encode(data: bytes, key: bytes):
    """Encode a byte array using the ECB mode and AES 128 encryption

    Args:
        data (bytes): plain text
        key (bytes): key for the AES

    Returns:
        bytes: AES 128 ECB encoded bytes
    """
    padded = pkcs7_pad(data, 128)
    return b"".join([aes128_encrypt(padded[i:i+BLOCKSIZE], key) for i in
range(0, len(padded), BLOCKSIZE)])


def ecb_decode(data: bytes, key: bytes):
    """Decode a byte array that was encoded using ECB and AES 128

    Args:
        data (bytes): encrypted data
```

```
        key (bytes): key for the AES

    Returns:
        bytes: the original plain text
    """
    padded = b"".join([aes128_decrypt(data[i:i+BLOCKSIZE], key) for i in
range(0, len(data), BLOCKSIZE)])
    return pkcs7_strip(padded)


def cbc_encode(data: bytes, key: bytes, iv: bytes):
    # pad to 16 bytes
    padded = pkcs7_pad(data, 128)

    # go one block at a time through the input and encrypt it
    ciphertext = []
    iv_temp = iv
    for i in range(0, len(padded), BLOCKSIZE):
        # xor the block with the iv
        data_xor_iv = bytes(a ^ b for a, b in zip(padded[i:i+BLOCKSIZE],
iv_temp))

        # update the iv
        iv_temp = aes128_encrypt(data_xor_iv, key)

        # add to the cipher text
        ciphertext.append(iv_temp)

    return b"".join(ciphertext)



def cbc_decode(data: bytes, key: bytes, iv: bytes):
    # deal with every block but the first
    plaintext = []
    for i in reversed(range(BLOCKSIZE, len(data), BLOCKSIZE)):
        # iv is the previous block
        iv_temp = data[i-BLOCKSIZE:i]

        # decrypt the cipher text, but we still need the xor
        needs_xor = aes128_decrypt(data[i:i+BLOCKSIZE], key)

        # convert to plaintext
        plaintext.append(bytes(a ^ b for a, b in zip(needs_xor, iv_temp)))
```

```
    # do the final block with the iv
    needs_xor = aes128_decrypt(data[:BLOCKSIZE], key)
    plaintext.append(bytes(a ^ b for a, b in zip(needs_xor, iv)))

    return pkcs7_strip(b"".join(reversed(plaintext)))
```

task1.py

```
import os
import os.path as osp
import sys
from block_cipher import (
    BLOCKSIZE, ecb_encode, cbc_encode
)

BMP_HEADER_LEN = 54


def main():
    # iterate through all arguments other than the python file name
    for file in sys.argv[1:]:
        # get a key and initialization vector
        key = os.urandom(BLOCKSIZE)
        iv = os.urandom(BLOCKSIZE)

        # if the argument isn't a file that exists
        if not osp.exists(file):
            print(f"{file} does not exist, skipping...")
            continue

        # get the name and the extension separately
        name, ext = osp.splitext(osp.basename(file))

        # if it is an image...
        if ext == ".bmp":
            # read the file into a bytes object
            with open(file, "rb") as f:
                contents = f.read()

            # split into header and data
            header, data = contents[:BMP_HEADER_LEN],
contents[BMP_HEADER_LEN:]

            # generate the ecb version
            ecb_data = ecb_encode(data, key)
            with open(f"output/{name}_ecb{ext}", "wb") as f:
                f.write(header)
                f.write(ecb_data)

            # generate the cbc version
            cbc_data = cbc_encode(data, key, iv)
            with open(f"output/{name}_cbc{ext}", "wb") as f:
```

```
                f.write(header)
                f.write(cbc_data)

        # every other type of file...
        else:
            # read the file into a bytes object
            with open(file, "rb") as f:
                data = f.read()

            # generate the ecb version
            ecb_data = ecb_encode(data, key)
            with open(f"output/{name}_ecb{ext}", "wb") as f:
                f.write(ecb_data)

            # generate the cbc version
            cbc_data = cbc_encode(data, key, iv)
            with open(f"output/{name}_cbc{ext}", "wb") as f:
                f.write(cbc_data)


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("usage: task1.py [file1] [file2] ...")
    main()
```

task2.py

```
import os
from block_cipher import cbc_encode, cbc_decode


def main():
    global KEY
    global IV

    # generate a random key and initialization vector
    KEY = os.urandom(16)
    IV = os.urandom(16)

    # set up a message, the zero block is on a block of its own
    eleven = "".join(["0" for _ in range(11)])
    zero_block =
"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    fifteen = "".join(["0" for _ in range(15)])

    # encrypt the message
    message = submit(eleven + zero_block + fifteen)

    # once we have the encrypted message, xor the block before the zero block
    # with the target string
    e_block = message[16:33]
    temp = b";admin=true....."
    new_block = bytes(b ^ a for a, b in zip(e_block, temp))
    new_message = message[:16] + new_block + message[32:]

    # verify the message
    print(verify(new_message))
    print(cbc_decode(new_message, KEY, IV))


def submit(s: str) -> bytes:
    """Prepends and appends some information to the given string
    as well as URL encoding `=` and `;`

    21 before, 17 after

    Args:
        s (str): string to use as userdata

    Returns:
        bytes: the encoded output of the string with extra information
```

```
    """
    s.replace(";", "%3B")
    s.replace("=", "%3D")
    new_s = "userid=456; userdata=" + s + ";session-id=31337"

    return cbc_encode(bytes(new_s, "utf8"), KEY, IV)


def verify(ciphertext: bytes) -> bool:
    """Checks to see if the string `;admin=true` is in an encoded message

    Args:
        ciphertext (bytes): encrypted text

    Returns:
        bool: whether or not `;admin=true` was found
    """
    plaintext = cbc_decode(ciphertext, KEY, IV)

    if b";admin=true" in plaintext:
        return True
    return False


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

task3.py

```
import pandas as pd
import matplotlib.pyplot as plt


def k_to_decimal(num: str):
    if isinstance(num, str) and num[-1] == 'k':
        return float(num[:-1]) * 1000
    else:
        return float(num)


def s_to_decimal(num: str):
    if isinstance(num, str) and num[-1] == 's':
        return float(num[:-1])
    else:
        return float(num)


def main():
    # generate the two csv files by running `openssl speed [type]`
    # once for aes -> aes.csv
    # once for rsa -> rsa.csv


    # read the inputs in
    with open('input/aes.csv', 'r') as f:
        aes_df = pd.read_csv(f)
    for col in aes_df.columns[1:]:
        aes_df[col] = aes_df[col].apply(k_to_decimal)

    with open('input/rsa.csv', 'r') as f:
        rsa_df = pd.read_csv(f)
    rsa_df['sign/s'] = rsa_df['sign/s'].apply(float)
    rsa_df['verify/s'] = rsa_df['verify/s'].apply(float)

    figure, axis = plt.subplots(1, 2)
    aes_axis, rsa_axis = axis

    # plot aes data
    for _, row in aes_df.iterrows():
        aes_axis.plot(row.values[1:], row.index[1:], label=row.values[0])
    aes_axis.set_xlabel('Throughput (bits/s)')
    aes_axis.set_ylabel('Block size')
    aes_axis.legend(bbox_to_anchor=(0,1.02,1,0.2), loc="lower left",
```

```
                    mode="expand", borderaxespad=0, ncol=3)

    # plot rsa data
    rsa_axis.plot('sign/s', 'type', data=rsa_df, label='sign')
    rsa_axis.plot('verify/s', 'type', data=rsa_df, label='verify')
    rsa_axis.set_xlabel('Throughput (bits/s)')
    rsa_axis.set_ylabel('Key size')
    rsa_axis.legend(bbox_to_anchor=(0,1.02,1,0.2), loc="lower left",
                    mode="expand", borderaxespad=0, ncol=3)

    plt.show()


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