Module RM3: Assignment 1 - Block Ciphers

Task 1: Modes of Operation

ECB: Break the plaintext into 128 bit (16B) blocks, then apply the same key to each block with AES, padding the final block if needed.
CBC: A 128 bit (16B) initialization vector (IV) is created and XORed with the first 128 bit (16B) block of plaintext. Apply AES to the first block, then XOR the result with the next block before encrypting to repeat the process again, padding the final block if needed.
PKCS#7 Padding: If N is the number of bytes needed to round a block to 16B, then N bytes of the value N are appended to the end of the block.

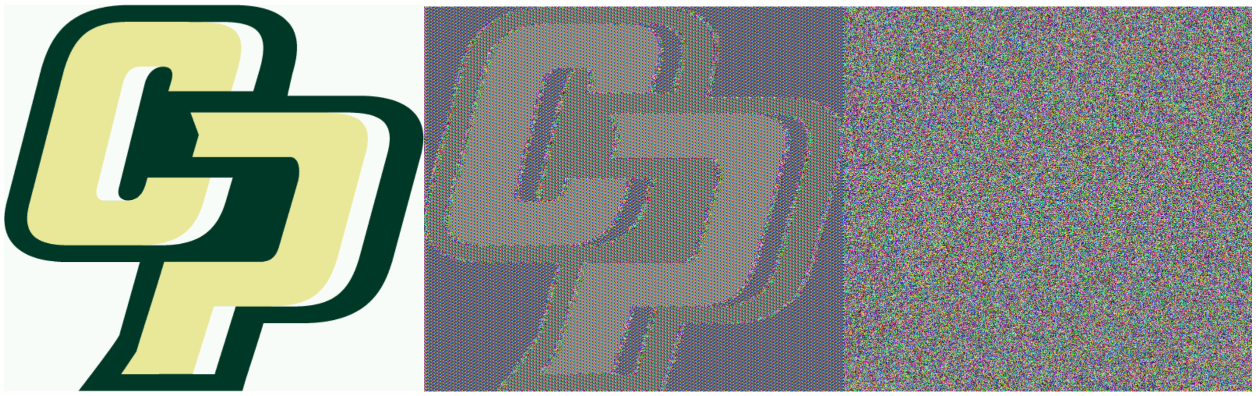


Figure 1: (From left to right) Cal Poly logo unmodified, encrypted with AES in ECB mode, and encrypted with AES in CBC mode

From the figure above, it is immediately apparent that while the exact data of the image is changed with ECB encryption, the outline of the logo is still very clear. This is because all blocks of 16 bytes with the same initial value will be encrypted to the same ciphertext. This weakness allows us to see areas of the image that share the same value, even if we don't know what that value is. Since CBC mode will produce two different ciphertexts for two identical adjacent blocks, this weakness is not present.

Code (some tabbing may have been lost):

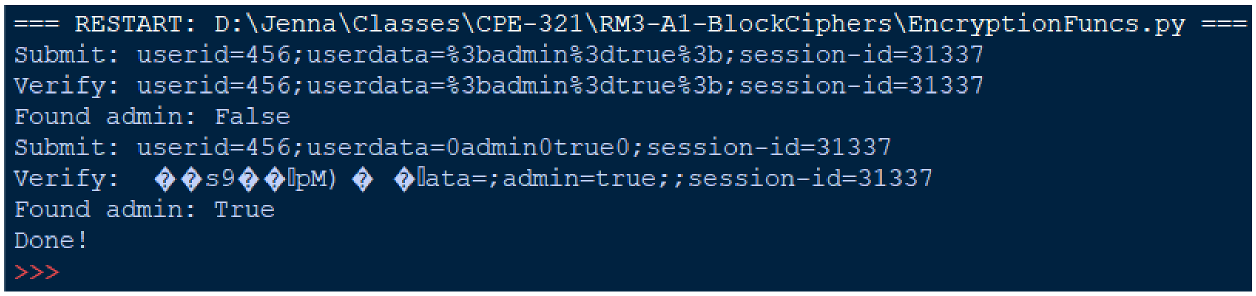
```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.strxor import strxor

class aes_encrypt:
      # Init
      def __init__(self):
      self.null_block =
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
      self.block_size = 16
      self.bmp_header_size = 54

      # Takes a bytes of length <= 16
      # Returns data padded with pkcs7
      def pad_data(self, data):
      # Check length
      if len(data) > 16:
            print("padded data was over 16B!")
            return self.null_block

      remainder = self.block_size - len(data)
      clean_data = data
      for i in range(remainder):
            clean_data += remainder.to_bytes(1, "little")

      return clean_data

      # Takes a bytes of max length 16 and encodes it with ecb
      # Returns the ciphertext as a bytearray
      def ecb_raw(self, data, key):
      # Note, a mode MUST be given, but only one block
      #  will be encoded at a time, so it has no effect
      cipher = AES.new(key, AES.MODE_ECB)

      # Pad data and encrypt
      clean_data = self.pad_data(data)
      ciphertext = cipher.encrypt(clean_data)

      return ciphertext

      # Takes 2 bytes objects of max length 16 and encodes the first with
```

```
cbc
        # Returns the ciphertext as a bytearray
        def cbc_raw(self, data, prev, key):
        # Check length
        if len(prev) != self.block_size:
                return self.null_block

        # pad data to length of prev
        clean_data = self.pad_data(data)

        # XOR data and prev
        final = strxor(clean_data, prev)
        ciphertext = self.ecb_raw(final, key)

        return ciphertext

        # Takes two strings of bmp file addrs to be read from and written to
        def encrypt_ecb(self, infile, outfile, bmp=False):
        # Setup
        f_in = open(infile, "rb")
        f_out = open(outfile, "wb")
        key = get_random_bytes(16)

        # Copy over header if this is a bmp file
        if bmp:
                header = f_in.read(self.bmp_header_size)
                f_out.write(header)

        # Encrypt blocks until done
        while True:
                # Read block
                data = f_in.read(self.block_size)
                # Terminate if EOF
                if len(data) == 0:
                        break
                # Process block
                ciphertext = self.ecb_raw(data, key)
                f_out.write(ciphertext)

        # Clean up
        f_in.close()
        f_out.close()
```

```python
    # Takes two strings of bmp file addrs to be read from and written to
    def encrypt_cbc(self, infile, outfile, bmp=False):
    # Setup
    f_in = open(infile, "rb")
    f_out = open(outfile, "wb")
    key = get_random_bytes(16)
    iv = get_random_bytes(16)

    # Copy over header if this is a bmp file
    if bmp:
        header = f_in.read(self.bmp_header_size)
        f_out.write(header)

    # Initialize prev with initialization vector
    prev = iv

    # Encrypt blocks until done
    while True:
        # Read block
        data = f_in.read(self.block_size)
        # Terminate if EOF
        if len(data) == 0:
            break
        # Process block
        ciphertext = self.cbc_raw(data, prev, key)
        f_out.write(ciphertext)
        # Update prev
        prev = ciphertext

    # Clean up
    f_in.close()
    f_out.close()

# Main program start
if __name__ == "__main__":
    # Setup
    ae = aes_encrypt()

    # Encrypt bmp files
    ae.encrypt_ecb("cp-logo.bmp", "out1.bmp", bmp=True)
    ae.encrypt_cbc("cp-logo.bmp", "out2.bmp", bmp=True)

    print("Done!")
```

Task 2: Limits of Confidentiality

By using the known user input "0admin0true0" and flipping the bits that overlap with the 0's in the previous block, we can have the program decrypt the string ";admin=true;" despite the submit function supposedly forbidding this. Since the character '0' has a value of 0x30, we can perform an XOR with the values 0x0b and 0x0d to get the characters ';' and '=' with values 0x3b and 0x3d. This is possible because the last step in decoding a block is XORing the decrypted data with the previous block's ciphertext. By the properties of XOR operations, intentionally toggling the bits of that block carries through and toggles the bits in the plaintext when the operation is repeated. The counter (CTR) mode of operation would make this more difficult, while still maintaining a block format. The counter mode XORs the block with a counter value that is encrypted, forming a block of seemingly random data that can still be undone. Now that blocks aren't XORed with each other, this form of attack is impossible.



```
=== RESTART: D:\Jenna\Classes\CPE-321\RM3-A1-BlockCiphers\EncryptionFuncs.py ===
Submit: userid=456;userdata=%3badmin%3dtrue%3b;session-id=31337
Verify: userid=456;userdata=%3badmin%3dtrue%3b;session-id=31337
Found admin: False
Submit: userid=456;userdata=0admin0true0;session-id=31337
Verify:  ��s9��▯pM) � �▯ata=;admin=true;;session-id=31337
Found admin: True
Done!
>>>
```

Figure 2: Console output from below program showing attack succeeds.

Code (some tabbing may have been lost):

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.strxor import strxor

class crypt_server:
	# Init
	def __init__(self):
	self.null_block =
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
	self.block_size = 16
	self.key = get_random_bytes(16)
	self.iv = get_random_bytes(16)

	# Takes a bytes of length <= 16
	# Returns data padded with pkcs7
	def pad_data(self, data):
        # Check length
        if len(data) > 16:
              print("padded data was over 16B!")
              return self.null_block

        remainder = self.block_size - len(data)
        clean_data = data
        for i in range(remainder):
              clean_data += remainder.to_bytes(1, "little")

        return clean_data

        # Takes 2 bytes objects of max length 16 and encodes the first with
  cbc
        # Returns the ciphertext as a bytearray
        def cbc_raw(self, data, prev):
        # Check length of prev
        if len(prev) != self.block_size:
              return self.null_block

        # Create AES cipher with key
        # Note, a mode MUST be given, but only one block
        #  will be encoded at a time, so it has no effect
        cipher = AES.new(self.key, AES.MODE_ECB)

        # pad data to length of prev
        clean_data = self.pad_data(data)
        # XOR data and prev
        final = strxor(clean_data, prev)
        # Create ciphertext from XORed data
        ciphertext = cipher.encrypt(final)

        return ciphertext

        # Takes two strings of bmp file addrs to be read from and written to
        def encrypt_string(self, val):
        # Setup
        val_bytes = bytes(val, 'utf-8')
        out = b''

        # Initialize prev with initialization vector
        prev = self.iv
```

```
# Encrypt blocks until done
i = 0;
while True:
      # Read block
      data = val_bytes[i:i+self.block_size]
      # Terminate if end of string
      if len(data) == 0:
            break
      # Process block
      ciphertext = self.cbc_raw(data, prev)
      out += ciphertext
      # Update prev
      prev = ciphertext
      # Increment counter
      i += self.block_size

return out

# Takes a string to be formatted and encoded
# Returns a bytes object of resulting ciphertext
def submit(self, val):
# Replace unsafe characters (; =) with safe encodings (%3b %3d)
clean_val = val.replace(";", "%3b")
clean_val = clean_val.replace("=", "%3d")

# Append and prepend extra data
clean_val = "userid=456;userdata="+clean_val+";session-id=31337"
print("Submit: "+clean_val)

# Encrypt
ciphertext = self.encrypt_string(clean_val)

return ciphertext

# Takes an encrypted bytes object to decrypt
# Returns True if decrypted ciphertext contains ";admin=true;"
def verify(self, val):
# Check length
if len(val) % self.block_size != 0:
      return False

# Setup
blocks = len(val) // self.block_size
```

```
        cipher = AES.new(self.key, AES.MODE_ECB)
        plain_blocks = [None]*blocks
        plaintext = ""

        # Do CBC backwards
        prev = b''
        while blocks > 0:
              # Get block
              data = val[self.block_size*(blocks-1):self.block_size*blocks]
              # Decrypt block
              data = cipher.decrypt(data)
              # Get previous block
              if blocks == 1:
                    # Use IV for last block
                    prev = self.iv
              else:
                    # Get block before current
                    prev =
  val[self.block_size*(blocks-2):self.block_size*(blocks-1)]
              # Undo XOR and save result
              plain_blocks[blocks-1] = strxor(prev,data)
              # Decrease block counter
              blocks -= 1;

        # Remove end padding through dark wizardry
        pad_val = plain_blocks[-1][self.block_size-1:]
        pad_len = plain_blocks[-1][-1]
        if (plain_blocks[-1][self.block_size-pad_len:self.block_size] ==
  pad_val*pad_len):
              plain_blocks[-1] = plain_blocks[-1][:self.block_size-pad_len]

        # Rebuild string from blocks
        for block in plain_blocks:
              plaintext += block.decode('utf-8', 'replace')

        # Check for ;admin=true;
        print("Verify: "+plaintext)
        return plaintext.find(";admin=true;") != -1

  # Main program
  if __name__ == "__main__":
        # Setup "server"
        server = crypt_server()
```

```
ret = server.submit(";admin=true;")
res = server.verify(ret)
print("Found admin: "+str(res)) # False

ret = server.submit("0admin0true0")
# These 3 0s can be modified by flipping bits
#  in the 5th, 11th, and 16th bytes of the ciphertext
# '0' = 0x30, ';' = 0x3b, '=' = 0x3d
ret_list = list(ret)
ret_list[4] ^= 0x0b
ret_list[10] ^= 0x0d
ret_list[15] ^= 0x0b
ret_modded = bytes(ret_list)
res = server.verify(ret_modded)
print("Found admin: "+str(res)) # True

print("Done!")
```

Task 3: Performance Comparison

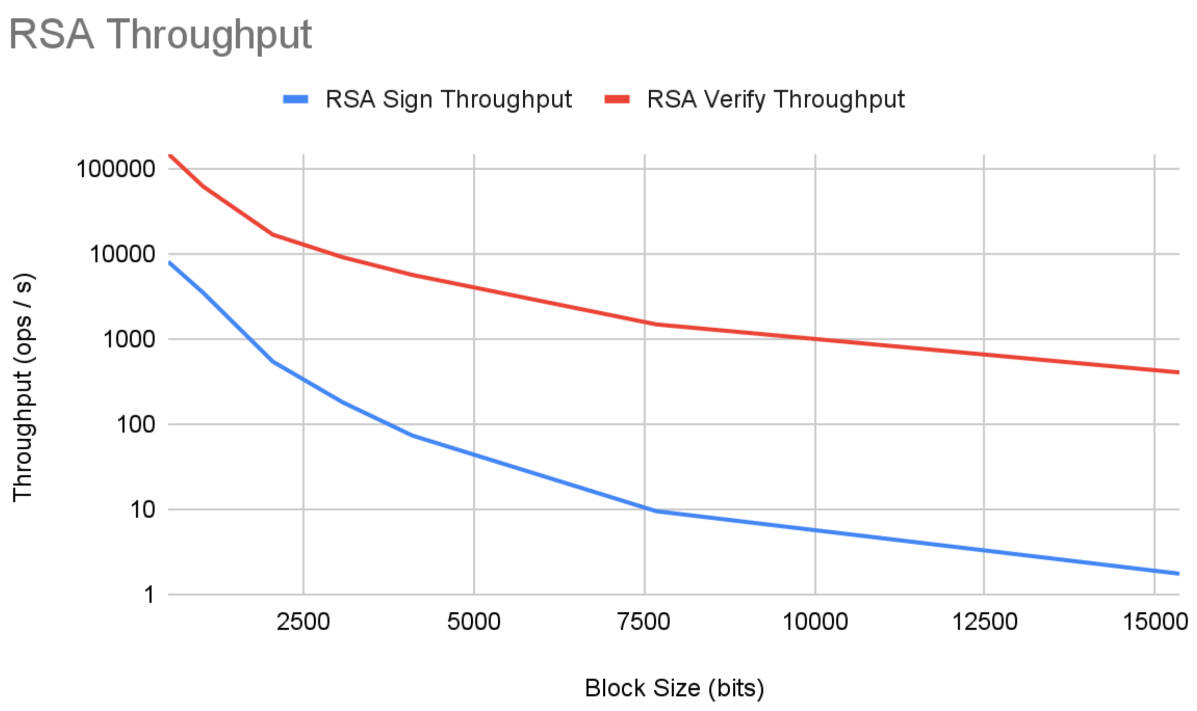


Figure 3: Graph of throughput vs block size for RSA signing and verifying operations

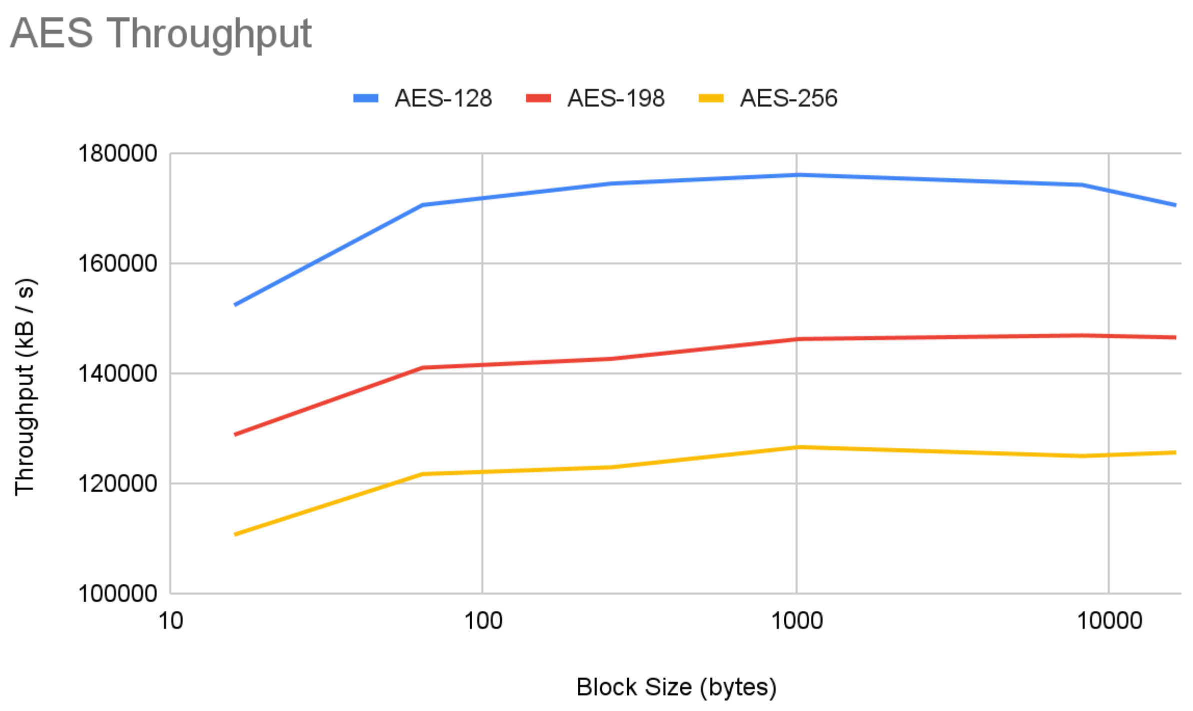


Figure 4: Graph of throughput vs block size for three different AES key lengths

The graph for RSA performance (Figure 3) was pretty close to expectations; a smaller block size is less data to process at once, so we expect each operation to take less time and increase throughput. We expected RSA verify operations to be slightly faster than sign operations, but we didn't expect it to be faster by over a factor of 10. As for the AES graph, we expected the trend of smaller key sizes having faster throughput, as that would follow the trend of RSA, but we didn't expect that throughput wouldn't strictly decrease with block size. Our graphs imply that a bock size around 1024 provides the highest possible throughput and much more or less that that decreases throughput. This pattern is clear for AES-128, but AES-198 and AES-256 might have slightly higher optimal block sizes.