MSCS-630 FINAL EXAM

Marcos Barbieri

PROFESSOR PABLO RIVAS MSCS-630

Question 1

Part A

At first glance, the student's code shows an incorrect process for iterating through rounds in the AES encryption process. The student correctly maintains a `roundCounter` variable to perform some logic on the `masterText` variable in the `generateCipher` method, but never increments the round. Thus, if left without incrementing the `roundCounter` value remains at 0. To correctly run the code and create the encrypted output an expression, incrementing the `roundCounter` variable, must be inserted in the branch that checks if the current `roundCounter` value is not the last round. This must be placed before an inner branch that performs another check to see if the `roundCounter` value is the last. The `roundCounter` increment expression, must be placed before said branch, because the *Mix Columns* step should not be executed in the last round, while the *Add Key, Nibble Substitution,* and *Shift Rows* should be executed in the last round.

Part B

A code snippet containing the corrected code from the `generateCipher` method in the `Aescipher` class can be found in the AES Cipher Fixed Code section of the Appendix of this document. Any changes made to the code are shown in green.

Question 2

Part A

From experience, it is clear that the code provided in the `Aescipher-buggy.java` can be optimized. One of the things that heavily increases the performance of the encryption process is the Galois Multiplication section. As of now the student decided to perform the calculation for the `multiply2` function. In addition, the `multiply3` function performs an XOR operation on the input and the input passed through the `multiply2` function, which retains the complexity of the `multiply2` function. Instead of performing the Galois Multiplication calculations in the code, we can use lookup tables to retrieve the values of the supposed input. This yields in a simpler method to get the Galois Multiplication and avoids having to compute the result in the code. In addition, we can make the code simpler (thereby faster) by replacing loops of a fixed size with static assignments.

Part B

The code that was changed can be found in the AES Cipher Optimized Code section of the Appendix. The code displayed merely represents snippets of the entire program. Anything in green was added to optimize the code, replacing anything in red, which was commented out. Further explanations as to why things were added/commented out can be found in comments above the statements of code (in their respective colors).

Question 3

The Advanced Encryption Standard (AES) is a successor to IBM's Data Encryption Standard (DES), made for encrypting sensitive government data. After a series of challenges that were performed to break DES succeeded, another challenge was made to create a better, more secure and efficient encryption standard. As of around 1998, the National Institute of Standards & Technology (NIST) had received 15 proposals for encryption standards. When the time came, five finalist proposals were examined, out of which AES was chosen as the new standard and officially adopted in 2001. Thus, AES evolved from all the brilliant cryptographers available, creating a more diverse set of proposals, as opposed to DES which was by a single entity, IBM. Even though IBM was a tech giant at the time, they only had their single perspective to test the DES algorithm, which only made the algorithm weaker. While this was an important step in the cryptography world, the process of finding a new proposed standard, after DES, was much more efficient, and has therefore surfaced a more secure cryptography standard. The success of AES arose from its fundamental process which consists of, an S-Box transformation on the key values; with 9 rounds for a 128-bit key, 11 rounds for a 192-bit key and 13 rounds for a 256-bit key. The encryption process then performs a shift row transformation, followed by a mix columns transformation, consisting of a Galois Multiplication over the matrix columns. Each round is concluded by XORing the round key with the AES plaintext at its state in each round. AES uses the Rijndael algorithm, developed by two Belgian cryptographers, Vincent Rijmen and Joan Daemen. The Rijndael algorithm was made to support key lengths of 128, 192, and 256 bits, making it far superior than the maximum 56-bit key length constraint on the algorithm used by DES. In addition, the Rijndael algorithm supports 128-bit block sizes as opposed to DES' 64-bit blocks.

Question 4

Mode	Advantages	Disadvantages
ECB [DO NOT USE]	- [Performance] Simple - [Performance] Easily encrypted and decrypted after modification - [Security] Modifications in in ciphertext are unpredictable, thus cannot predict a certain change plaintext - [Performance] Encryption and decryption can be parallelized	- [Security] Equal plaintext blocks can yield equal ciphertext blocks which can be a problem - [Security] Lack of diffusion makes it easy to find patterns in encrypted data
CBC	- [Security] Same input, but different encryption instance can yield different output (decreased transparency) - [Security] Message can be decrypted from any section - [Performance] Adjacent blocks can be used to decrypt, thus decryption can be parallelized	- [Performance] Encryption process is sequential, thus cannot be parallelized - [Performance] Have to re-encrypt after modifying since each block uses previous block
CBC fixed IV [DO NOT USE]	-	- [Security] Can be exploited by a chosen plaintext attack - [Security] Attacker can detect if two different messages start with the same plaintext block
CBC counter IV	- [Security] Counter creates a more diffused initialization vector	- [Security] Too small of a difference in the ciphertext block if you increment counter by 1, thus attacker will detect the differences quickly
CBC random IV	- [Security] Defends against chosen plaintext attacks by having a random initialization vector	- [Security] Recipient of message needs to know IV - [Performance] Ciphertext is one block longer since you have to send random IV

Question 5

Suppose we have a system within a company that serves data to a client UI for certain employees working on a government contract. Typically, projects involving government contracts contain sensitive data, which must be thoroughly encrypted for transmission. In the proposed system, we have a server with a database containing sensitive information, and the employees are able to see that information only by logging into a specific portal. It is highly likely that in order to transmit this information to the portal, authentication will be required; in this case suppose we are using CBC-MAC. Now, in this scenario it is likely that not only CBC-MAC be used for authentication, but for encryption as well. One of the vulnerabilities of CBC-MAC lies in using the same key for authentication and encryption; this assumes that the initialization vector is an array of zeroes. By using the same key for authentication and encryption, it is possible that an attacker may change some bits while the ciphertext is in transit, and the decryption process may generate the same authentication code even though the message is different. Thus, by using a different key for each process, if this were to play out similarly, the receiving end will not be able to decrypt the message and will therefore understand that something has changed. In addition, by using a predictable initialization vector an attacker can examine past messages sent and retrieve an initialization vector that will generate a valid MAC. Thus, it is key that random initialization vectors be used as frequently as possible.

Question 6

Using the Website Security Summary Table in the appendix, we can see some interesting trends in web security. The first thing to note is that all websites checked use AES. Since AES is the dominant technology for encryption it is good that every website uses AES. Thus, it is unlikely that any website will be using DES or 3DES for their encryption mechanisms. Another very notable trend, is the overwhelming use of the 128-bit key size for AES. Only one website opened on tab #6 used a key size of 256. This is because AES with a key size of 128 bits is strong enough to protect any data being transmitted, and has a serious performance boost compared to the use of AES 256. Using AES with a 256-bit key, while more secure can slow down the transmission of data, which could seriously affect customer satisfaction regarding the company's online presence. The performance degradation was actually pretty noticeable in tab #6, but would be premature to blame the encryption mechanism for the apparent latency of the website. Nevertheless, if performance/speed is something that is important for the use of the website, it is clear that AES with a 128-bit key is sufficiently secure and low-latency to meet the needs of the users. In addition, all websites opened used AES in Galois/Counter Mode (GCM). This is likely to be because GCM is extremely fast, patent-free, and can be parallelized which makes it extremely suitable for any cloud applications. Thus, it is not obscene to see that 100% of the websites tested in this used AES in Galois/Counter Mode (GCM).

Appendix

AES Cipher Fixed Code

Aescipher.java

```
while (WCol < column_size) {
  for (int cols = 0; cols < 4; cols++, WCol++) {
    for (int row = 0; row < 4; row++) {
      keyHex[row][cols] = keyMatrixW_encrypt[row][WCol];
    }
  }

  if (roundCounter != (rounds - 1)) {
    // added increment to roundCounter so the code executes
    // correct logic for each round
    roundCounter++;
    masterText = aesStateXor(masterText, keyHex);
    masterText = aesNibbleSub(masterText);
    masterText = aesShiftRow(masterText);
    if (roundCounter != (rounds - 1)) {
      masterText = aesMixColumn(masterText);
    }

  } else
    masterText = aesStateXor(masterText, keyHex);
}
```

AES Cipher Optimized Code

Aescipher.java

```
public static String generateCipher(String[][] masterKey,
String[][] masterText, int column_size, int row_size,
      int rounds) {

    String[][] keyHex = new String[4][4];
    String out = "";
    int WCol = 0;
    int roundCounter = 0;
    while (WCol < column_size) {
      for (int cols = 0; cols < 4; cols++, WCol++) {
          // do this manually instead of having to use
          // a for-loop
          keyHex[0][cols] = keyMatrixW_encrypt[0][WCol];
          keyHex[1][cols] = keyMatrixW_encrypt[1][WCol];
          keyHex[2][cols] = keyMatrixW_encrypt[2][WCol];
          keyHex[3][cols] = keyMatrixW_encrypt[3][WCol];
          /*
          // this was how it was done before don't need for-loop
```

```
            for (int row = 0; row < 4; row++) {
              keyHex[row][cols] = keyMatrixW_encrypt[row][WCol];
            }*
            /
        }

        if (roundCounter != (rounds - 1)) {
          roundCounter++;
          masterText = aesStateXor(masterText, keyHex);
          // Exclusive or output is passed to nibble substitution
is
          // called
          masterText = aesNibbleSub(masterText);
          // Nibble substituted output is called to shiftrows
method
          masterText = aesShiftRow(masterText);
          // Shifted output is sent to mixing columns function
          if (roundCounter != (rounds - 1)) {
            masterText = aesMixColumn(masterText);
          }

        } else
          // In the tenth round we do only plain xor
          masterText = aesStateXor(masterText, keyHex);
      }
      // System.out.println("The Cipher Text is");
      for (int cols = 0; cols < 4; cols++) {
        for (int row = 0; row < 4; row++) {
          //outValue = outValue.append(masterText[row][cols]);
          out += masterText[row][cols];
          // System.out.print(masterText[row][cols]+ "\t");
        }

      }
      //System.out.println();
      // Aesdecipher.processInput(outValue, inputkey,
size_basket);
      return out;

    }

// adding lookup table for galois multiplication 0x02
private static final char[] mult2Lookup = {
      0x00,0x02,0x04,0x06,0x08,0x0a,0x0c,0x0e,
      0x10,0x12,0x14,0x16,0x18,0x1a,0x1c,0x1e,
      0x20,0x22,0x24,0x26,0x28,0x2a,0x2c,0x2e,
      0x30,0x32,0x34,0x36,0x38,0x3a,0x3c,0x3e,
```

```
    0x40,0x42,0x44,0x46,0x48,0x4a,0x4c,0x4e,
    0x50,0x52,0x54,0x56,0x58,0x5a,0x5c,0x5e,
    0x60,0x62,0x64,0x66,0x68,0x6a,0x6c,0x6e,
    0x70,0x72,0x74,0x76,0x78,0x7a,0x7c,0x7e,
    0x80,0x82,0x84,0x86,0x88,0x8a,0x8c,0x8e,
    0x90,0x92,0x94,0x96,0x98,0x9a,0x9c,0x9e,
    0xa0,0xa2,0xa4,0xa6,0xa8,0xaa,0xac,0xae,
    0xb0,0xb2,0xb4,0xb6,0xb8,0xba,0xbc,0xbe,
    0xc0,0xc2,0xc4,0xc6,0xc8,0xca,0xcc,0xce,
    0xd0,0xd2,0xd4,0xd6,0xd8,0xda,0xdc,0xde,
    0xe0,0xe2,0xe4,0xe6,0xe8,0xea,0xec,0xee,
    0xf0,0xf2,0xf4,0xf6,0xf8,0xfa,0xfc,0xfe,
    0x1b,0x19,0x1f,0x1d,0x13,0x11,0x17,0x15,
    0x0b,0x09,0x0f,0x0d,0x03,0x01,0x07,0x05,
    0x3b,0x39,0x3f,0x3d,0x33,0x31,0x37,0x35,
    0x2b,0x29,0x2f,0x2d,0x23,0x21,0x27,0x25,
    0x5b,0x59,0x5f,0x5d,0x53,0x51,0x57,0x55,
    0x4b,0x49,0x4f,0x4d,0x43,0x41,0x47,0x45,
    0x7b,0x79,0x7f,0x7d,0x73,0x71,0x77,0x75,
    0x6b,0x69,0x6f,0x6d,0x63,0x61,0x67,0x65,
    0x9b,0x99,0x9f,0x9d,0x93,0x91,0x97,0x95,
    0x8b,0x89,0x8f,0x8d,0x83,0x81,0x87,0x85,
    0xbb,0xb9,0xbf,0xbd,0xb3,0xb1,0xb7,0xb5,
    0xab,0xa9,0xaf,0xad,0xa3,0xa1,0xa7,0xa5,
    0xdb,0xd9,0xdf,0xdd,0xd3,0xd1,0xd7,0xd5,
    0xcb,0xc9,0xcf,0xcd,0xc3,0xc1,0xc7,0xc5,
    0xfb,0xf9,0xff,0xfd,0xf3,0xf1,0xf7,0xf5,
    0xeb,0xe9,0xef,0xed,0xe3,0xe1,0xe7,0xe5
  };

// adding lookup table for galois multiplication 0x03
private static final char[] mult3Lookup = {
    0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,
    0x18,0x1b,0x1e,0x1d,0x14,0x17,0x12,0x11,
    0x30,0x33,0x36,0x35,0x3c,0x3f,0x3a,0x39,
    0x28,0x2b,0x2e,0x2d,0x24,0x27,0x22,0x21,
    0x60,0x63,0x66,0x65,0x6c,0x6f,0x6a,0x69,
    0x78,0x7b,0x7e,0x7d,0x74,0x77,0x72,0x71,
    0x50,0x53,0x56,0x55,0x5c,0x5f,0x5a,0x59,
    0x48,0x4b,0x4e,0x4d,0x44,0x47,0x42,0x41,
    0xc0,0xc3,0xc6,0xc5,0xcc,0xcf,0xca,0xc9,
    0xd8,0xdb,0xde,0xdd,0xd4,0xd7,0xd2,0xd1,
    0xf0,0xf3,0xf6,0xf5,0xfc,0xff,0xfa,0xf9,
    0xe8,0xeb,0xee,0xed,0xe4,0xe7,0xe2,0xe1,
    0xa0,0xa3,0xa6,0xa5,0xac,0xaf,0xaa,0xa9,
    0xb8,0xbb,0xbe,0xbd,0xb4,0xb7,0xb2,0xb1,
    0x90,0x93,0x96,0x95,0x9c,0x9f,0x9a,0x99,
```

```
    0x88,0x8b,0x8e,0x8d,0x84,0x87,0x82,0x81,
    0x9b,0x98,0x9d,0x9e,0x97,0x94,0x91,0x92,
    0x83,0x80,0x85,0x86,0x8f,0x8c,0x89,0x8a,
    0xab,0xa8,0xad,0xae,0xa7,0xa4,0xa1,0xa2,
    0xb3,0xb0,0xb5,0xb6,0xbf,0xbc,0xb9,0xba,
    0xfb,0xf8,0xfd,0xfe,0xf7,0xf4,0xf1,0xf2,
    0xe3,0xe0,0xe5,0xe6,0xef,0xec,0xe9,0xea,
    0xcb,0xc8,0xcd,0xce,0xc7,0xc4,0xc1,0xc2,
    0xd3,0xd0,0xd5,0xd6,0xdf,0xdc,0xd9,0xda,
    0x5b,0x58,0x5d,0x5e,0x57,0x54,0x51,0x52,
    0x43,0x40,0x45,0x46,0x4f,0x4c,0x49,0x4a,
    0x6b,0x68,0x6d,0x6e,0x67,0x64,0x61,0x62,
    0x73,0x70,0x75,0x76,0x7f,0x7c,0x79,0x7a,
    0x3b,0x38,0x3d,0x3e,0x37,0x34,0x31,0x32,
    0x23,0x20,0x25,0x26,0x2f,0x2c,0x29,0x2a,
    0x0b,0x08,0x0d,0x0e,0x07,0x04,0x01,0x02,
    0x13,0x10,0x15,0x16,0x1f,0x1c,0x19,0x1a
  };

/**
   * In this function , mix columns operations having
multiplication
   * with 3 are considered here and operation is performed.
   *
   * @param InputHex
   * @return
   */
  protected static String multiply3(String InputHex) {
    // instead of calculating the multiplication by 3 we can use
    // a lookup table to improve performance
    return mult3Lookup[Integer.parseInt(InputHex, 16)] + "";
    // high latency way of doing it
    /*
    return exclusiveOr(multiply2(InputHex), InputHex);
    */
  }

  /**
   * In Mix columns operation if the element is to be multiplied
with 2, we
   * will use this function to perform the operation of checking
most
   * significant bit shifting the bits
   *
   * @param InputHex
   * @return
   */
```

```
  protected static String multiply2(String InputHex) {
    // instead of calculating the multiplication by 2 we can use
    // a lookup table to improve performance
    return mult2Lookup[Integer.parseInt(InputHex, 16)] + "";
    // high-latency way of doing it
    /*
    InputHex = Integer.toBinaryString(Integer.parseInt(InputHex,
16));
    int lenthOfInput = 8 - (InputHex.length());
    String pads = new String();
    for (int i = 0; i < lenthOfInput; i++) {
      pads += "0";
    }

    String Input = pads.concat(InputHex);
    String oneB = Integer.toHexString(27);
    String shiftedBinary =
Integer.toBinaryString(Integer.parseInt(Input, 2) << 1);
    if (shiftedBinary.length() > 8) {
      shiftedBinary = shiftedBinary.substring(1);
    }
    String shifted =
Integer.toHexString(Integer.parseInt(shiftedBinary, 2));

    if (Input.substring(0, 1).equals("1")) {
      return exclusiveOr(shifted, oneB);
    } else
    return shifted;
    */
  }
```

Website Security Summary Table

TAB #	SITE NAME	ENCRYPTION	KEY SIZE	MODE
1	github.com	AES	128	GCM
2	fbi.gov	AES	128	GCM
3	joelonsoftware.com	AES	128	GCM
4	stackoverflow.com	AES	128	GCM
5	stopandshop.com	AES	128	GCM
6	macys.com	AES	256	GCM
7	ilearn.marist.edu/portal	AES	128	GCM