Final Exam Writeup

Miguel Nistal

(1) Fix Broken AESCipher

The problem in the buggy AESCipher code was due to the encryption rounds found on line 111 and used between lines 119 and 133 (which are shown below):

```
    if (roundCounter != (rounds - 1)) {
        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);
```

I reimplemented this section adding the missing increment and re-arranging the rounds to correctly do the xor-only step as part of round 0:

```
    if (roundCounter == 0) {
        // 0th round is only an xor statement
        masterText = aesStateXor(masterText, keyHex);
    }
    else {
        masterText = aesNibbleSub(masterText);
        masterText = aesShiftRow(masterText);
        // In round 10, don't do last mix columns
        if (roundCounter != (rounds - 1)) {
            masterText = aesMixColumn(masterText);
        }
        masterText = aesStateXor(masterText, keyHex);
    }
    // Make sure you advance the round counter!!
    roundCounter++;
```

Replacing the top snippet of code with the bottom snippet causes the driver to produce the correct output without penalty.

(2) Improve Performance

The first clear performance improving change is converting the whole algorithm from operating on Strings to operating on integers. Converting from string to integer and back is costly and unnecessary. Since the entire algorithm only operates on bytes, the code should convert the input from the Strings provided by the input to an integer matrix and do all the operations on integers throughout, only performing another conversion at the end to print the result. Another benefit of this is the elimination of the exlusive or function which exemplifies why the algorithm shouldn't be implemented in strings - the entire function exists

to convert string to integer, do the builtin xor operation, and convert back to strings. When using integers directly - I was able to eliminate that function entirely and use the builtin operator.

The second performance change is to use lookup tables for the Galois multiplication. Doing the operations directly saves a little on memory but costs more in performance where the lookups will always happen in constant time. In addition to using the lookup tables for Galois multiplication, I removed the function calls that indexed to the other tables (i.e. SBOX and RCON), instead using the indexes directly inline. This eliminates a function call that isn't necessary (especially since we're just using integers directly now).

While doing the analysis of the results, I noticed quite a bit of variance despite the 100 repetitions in the timeit loop, which caused me to increase the number of repetions by a factor of 10 to try and get more stable results. With 100 repetitions I saw ~8-16% performance improvement, while with 1000 repetitions I saw a more consistent ~10%.

See the appendix for the updated code, with the comments *PERF* for comments indicating what was changed to improve performance.

(3) Encryption Standards

AES (known as the Rijndael cipher) was developed by Vincent Rijmen and Joan Daemen and supports 128, 192, and 256 bit key sizes, for a 128 bit block size. This algorithm was ultimately the replacement for DES, the original federal encryption standard for the United States. DES was the result of a research project set up by IBM in the 1960s which resulted in a cipher known as LUCIFER. In the 1970s, IBM worked with the NSA and others to commericialize the algorithm and the resultant algorithm was put forward as a proposal for a new encryption standard requested by the National Bureau of Standards. It was officially adopted in 1977 as the Data Encryption Standard. This closed doors process that was used to develop DES between IBM and the NSA was classified and scrutinized by many experts even today. The key size was reduced from 128 bits to 56 bits and the SBox was developed in secrecy. These changes made external experts believe that the NSA was intentionally watering down the algorithm so they could use a trap-door or brute force attack of their own design. In 1999 DES was publically broken in 22 hours bythe Electronic Frontier Foundation.

In 1997 NIST announced they wanted to choose a successor to DES known as AES and would be done by an open selection process chosen from algorithms designed by the cryptographic community. This open process lead to a much less controversial algorithm than DES was originally. This process continued for 3 years until the winner was announced after several rounds of reviews, the Rijndael cipher in fall of 2000. A year later, AES was officially approved as the new encryption standard.

(4) Block Cipher Modes

Two of the most common block cipher modes of operation are CBC (Cipher Block Chaining) and ECB (Electronic Code Book). The main disadvantage of Electronic Code Book mode is a lack of diffusion. Each plaintext block is encrypted entirely on its own so a repetitive pattern of bytes that's being encrypted still results in a repetitive pattern of bytes since each block will be encrypted in the same way. Electronic Code Book mode does outperform CBC mode, but only because it performs no additional operations. The extra chaining that CBC does adds an extra xor of the keymatrix for each block which isn't free, but is largely negligible considering the security advantages. The block chaining performed by the cipher in CBC mode homogenizes any recognizable patterns to make the result look much more noisy and less vunlerable to pattern recognition.

In order to do the first block of encryption in CBC mode, the algorithm requires an Initialization Vector (IV), to act as the first "previous block" of text to XOR with. Several examples of how to pick an IV are to use a Fixed IV, a Counter as an IV, or a pseudo-random IV. A fixed IV has the problem that any plaintext whose first block happens to align with the IV results in zero due to the nature of the xor. That means that any attacker who notices a ciphertext block of all zeroes means they know the first plaintext block was equal to the initialization vector. With a fixed IV, this scenario is bound to happen. Using a counter as the initialization vector can be just as bad. When using a predictable IV, an attacker can use a Chosen Plaintext Attack by submitting a suitable message to be encrypted with that IV. In this case the attacker can verify guesses at the contents of any ciphertext block seen before. Therefore due to the nature of XOR, it's very important that initialization vectors are chosen randomly, which minimizes both their collision with plaintext blocks, and the weakness against chosen ciphertext attacks.

(5) Message Authentication

Imagine I designed a new application designed to compete directly with Whatsapp. The goal would be to take aim at supposedly secure communication that was owned by coporations like Facebook and build it on open source software to ensure the integrity of the design was kept in tact despite growing popularity. In this system I could use CBC-MAC to authenticate that messages that were transferred between two devices aren't tampered with. When a sender creates a new message, I would use CBC-MAC to produce a MAC for the plaintext the sender has created, then encrypt the message, and send both to the reciever end. On the reciever end, the user's application would decrypt the message (using a secure cryptosystem based on some secure key

exchange protocol). Once the message was decrypted, the reciever's device would use CBC-MAC to produce the MAC for the recieved plaintext and compare the two. If the two codes are identical, then the message is safe to open and would be displayed to the user.

Naturally, messaging apps consist of variable length messages, so under this system we should be concerned about the security of CBC-MAC, since an attacker who knows the correct message tag pairs for two messages could perform a collision attack to send a different message in it's place. The easiest approach to rectifying this vulnerability is encrypting the last block of CBC. The other main concern is with what key to use for the CBC-MAC. Since I'm encrypting both the message and the MAC (CBC-MAC using a secure block cipher will require a key), I will need two keys. A simple, yet inelegant, solution to this could be to take the users password, salt/hash/expand it for the message encryption, and then use the password plaintext backwards, salt/hash/expand it for the CBC-MAC.

(6) Web Encryption

Website	Key Exchange	Cipher
Battle.net	ECDHE_RSA	AES128_GCM
Wikipedia	ECDHE_ECDSA	CHACHA20_POLY1305
Hearthpwn	ECDHE_ECDSA	AES128_GCM
Mmo-champion	ECDHE_ECDSA	AES128_GCM
Reddit	ECDHE_RSA	AES128_GCM
Spotify	ECDHE_RSA	AES128_GCM
Amazon	ECDHE_RSA	AES128_GCM

In the seven websites I visited, I found two key exchange protocols, and two ciphers used. The key exchange protocols utilized the same key aggrement protocol to establish a shared secret (Elliptic-curve Diffie-Hellman Ephemeral) but used two different cryptosystems in its implementation. The trade off here between ECDSA (Ecliptic Curve Digital Signature Algorithm) vs RSA (Rivest-Shamir-Adleman) is a question of size vs speed. It's much quicker to verify an RSA signature than an ECDSA signature, but the signature in ECDSA can be expressed in less than half the number of bytes as an equally secure RSA signature. Considering modern internet speeds I would probably prefer RSA as the increased size is a worthwhile tradeoff for speed (considering modern internet bandwidth), but it seems to be a negligible tradeoff in either direction which is supported by the nearly 50/50 split of websites using both.

In all seven websites though, only one used a cipher other than AES128 for it's encryption. Wikipedia is using CHACHA20 as it's cipher, which appears to be a competitor to AES invented by Daniel J Bernstein. It seems like it's being advertised as faster than AES without dedicated hardware. However, with Intel and AMD both producing products with AES accelerators lately, it seems unlikely for CHACHA20 to outperform AES128. Critics of AES will say this leaves it open to side-channel attacks based on hardware implementation knowledge, but ultimately for websites this seems like a low-risk possibility considering the effect of network latency greatly overshadows cache latency. Finally, CHACHA20 is advertised as being designed with AEAD (Authenticated Encryption with Acompanying Data), which gives the benefit of authenticity and security all in one. Considering the algorithm is still relatively new compared to AES, it's not unsuprising the only place I saw it's use was in the non-profit, Wikipedia.

Appendix (Aescipher.java)

Updated code for question (2) performance optimizations

```
/**
 * File: Aescipher.java
 *
 * It accepts user input, key and decrypts
 *         the cipher
 *
 */

public class Aescipher {
```

```
/* *PERF* Declare all the lookup tables in static memory, faster than doing actual math */
public static int SBOX[] = {
  0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
  0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
  0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
  0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
  0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
  0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
  0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
  0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
  0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
  0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
  0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
  0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
  0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
  0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
  0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
  0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
};

public static int RCON[] = {
  0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
  0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,
  0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
  0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
  0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
  0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
  0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
  0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
  0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
  0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
  0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
  0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
  0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
  0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,
  0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
  0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d
};

public static int GMUL_BY2[] = {
  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
  };

  public static int GMUL_BY3[] = {
    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
  };


  // State Matrices
  /* *PERF* Everything should be done as integers, strings are expensive */
  public static int[][] masterKey_encrypt;
  public static int[][] masterText_encrypt;
  public static int[][] keyMatrixW_encrypt;

  public static String processInput(String plainText, String inputKey, int[] size_basket, String verbose)
{
    int i = 0;
    int j = 0;
    int col_valueforInput;
    int column_size;
    int rounds;

    // Expand arguments - This is such a weird way to do this
    col_valueforInput = size_basket[0];
    column_size = size_basket[1];
    rounds = size_basket[2];

    /* *PERF* These are all integer matrices now */
    keyMatrixW_encrypt = new int[4][column_size];
    masterKey_encrypt = new int[4][col_valueforInput];
    masterText_encrypt = new int[4][4];

    for (int column = 0; column < col_valueforInput; column++) {
      for (int row = 0; row < 4; row = row + 1) {
                                        /* *PERF* Hex is a representation, deal with the bytes directly
*/
        masterKey_encrypt[row][column] = Integer.parseInt(inputKey.substring(i, i + 2), 16);
        i = i + 2;
```

```
      }
    }

    for (int column = 0; column < 4; column++) {
      for (int row = 0; row < 4; row = row + 1) {
                                        /* *PERF* Hex is a representation, deal with the bytes directly
*/
        masterText_encrypt[row][column] = Integer.parseInt(plainText.substring(j, j + 2), 16);
        j = j + 2;
      }
    }

    if (verbose.equals("1")) {
      System.out.println("Text to be encrypted after padding is");
      System.out.println(plainText);
    }

    /* *PERF* Don't store to locals just to leave them on the stack doing nothing */
    return generateWMatrix(col_valueforInput, column_size, rounds);
  }

  /** *NOTE* I moved this so the code reads in calling order, makes it easier to debug
   * generateWMatrix() method starts processing the keys for the 4*44 keys
   * matrix
   */
  public static String generateWMatrix(int col_valueforInput, int column_size, int rounds) {
    for (int row = 0; row < 4; row = row + 1) {
      for (int column = 0; column < col_valueforInput; column++) {
        keyMatrixW_encrypt[row][column] = masterKey_encrypt[row][column];
      }
    }

    // Processing the rest keys for keyMatrixW , by taking an intermediate
    // matrix wNewMatrix for processing purpose
    /* *PERF* More integer conversions */
    int[][] wNewMatrix = null;
    for (int column = col_valueforInput; column < column_size; column++) {
      /**
       * if the column number is not a multiple of 4 the following steps
       * are to be implemented
       */

      if (column % col_valueforInput != 0 && col_valueforInput == 8) {
        if (column % 4 == 0) {
          for (int row = 0; row < 4; row++) {
                                        /* *PERF* Don't waste a function call, just do a lookup! */
            keyMatrixW_encrypt[row][column] = SBOX[keyMatrixW_encrypt[row][column - 1]];
                                        /* *PERF* Got rid of the string xor */
            keyMatrixW_encrypt[row][column] = keyMatrixW_encrypt[row][column - col_valueforInput] ^
keyMatrixW_encrypt[row][column];

          }
        }
        else {
```

```
          for (int row = 0; row < 4; row++) {
                                        /* *PERF* Got rid of the string xor */
            keyMatrixW_encrypt[row][column] = keyMatrixW_encrypt[row][column - col_valueforInput] ^
keyMatrixW_encrypt[row][column - 1];
          }
        }
      }
      else if (column % col_valueforInput != 0 && col_valueforInput != 8) {
        for (int row = 0; row < 4; row++) {
                                        /* *PERF* Got rid of the string xor */
          keyMatrixW_encrypt[row][column] = keyMatrixW_encrypt[row][column - col_valueforInput] ^
keyMatrixW_encrypt[row][column - 1];
        }

      }
      else if (column % col_valueforInput == 0) {

        // If its a multiple of 4 the following steps will be
        // implemented
        wNewMatrix = new int[1][4];
        // Inserting values and shifting cells in the intermediate
        // matrix
        wNewMatrix[0][0] = keyMatrixW_encrypt[1][column - 1];
        wNewMatrix[0][1] = keyMatrixW_encrypt[2][column - 1];
        wNewMatrix[0][2] = keyMatrixW_encrypt[3][column - 1];
        wNewMatrix[0][3] = keyMatrixW_encrypt[0][column - 1];

        // Once the shifting is done we do the s-box transformation
        for (int i = 0; i < 1; i++) {
          for (int j = 0; j < 4; j++) {
                              /* *PERF* Replaced func with lookup */
            wNewMatrix[i][j] = SBOX[wNewMatrix[i][j]];
          }
        }

        int r = column / col_valueforInput;
        // Performing XOR of the R_CON value and new matrix value
                            /* *PERF* Replaced func with Lookup */
                            /* *PERF* Got rid of the string xor */
        wNewMatrix[0][0] = RCON[r] ^ wNewMatrix[0][0];

        // Final computation of recursively XOR the values
        // - I don't think this is recursion
        for (int row = 0; row < 4; row++) {
                            /* *PERF* Got rid of the string xor */
          keyMatrixW_encrypt[row][column] = keyMatrixW_encrypt[row][column - col_valueforInput] ^
wNewMatrix[0][row];
        }
      }
    }

    /* *PERF* Don't store to locals just to leave them on the stack doing nothing */
    return generateCipher(masterKey_encrypt, masterText_encrypt, column_size, col_valueforInput, rounds);
  }
```

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

    //*TODO* LEFT OFF HERE!!!!!!!!!!
    int[][] keyHex = new int[4][4];
    StringBuilder outValue = new StringBuilder();
    int WCol = 0;
    int roundCounter = 0;
    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 == 0) {
        masterText = aesStateXor(masterText, keyHex);
      }
      else {
        masterText = aesNibbleSub(masterText);
        masterText = aesShiftRow(masterText);
        if (roundCounter != (rounds - 1)) {
          /* *PERF* this function looks different because I dropped in a lookup table implementation to
replace it */
          masterText = AESMixColumns(masterText);
        }
        masterText = aesStateXor(masterText, keyHex);
      }
      roundCounter++;
    }
    // System.out.println("The Cipher Text is");
    for (int cols = 0; cols < 4; cols++) {
      for (int row = 0; row < 4; row++) {
        outValue = outValue.append(String.format("%02X", masterText[row][cols]));
        // System.out.print(masterText[row][cols]+ "\t");
      }

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

  }

  /* *PERF* Removed unnecessary String XOR function to ensure its not used anywhere else */
  /* *PERF* Removed unecessary SBOX function to ensure its not used anywhere */
  /* *PERF* Removed RCON too */

  public static int[][] aesStateXor(int[][] sHex, int[][] keyHex) {
    int exclusiveOrArray[][] = new int[4][4];
    for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4; j++) {
                    /* *PERF* replaced xor function call */
```

```
      exclusiveOrArray[i][j] = sHex[i][j] ^ keyHex[i][j];
    }
  }

  return exclusiveOrArray;
}

/**
 * Accepts Exclusiveor output and finds the respective element in S_BOX
 * matrix
 *
 * @param exclusive
 * @return
 */
public static int[][] aesNibbleSub(int[][] exclusive) {
  int sBoxValues[][] = new int[4][4];
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
      sBoxValues[i][j] = SBOX[exclusive[i][j]];
    }
  }
  return sBoxValues;
}

/**
 * Once the S_BOX values are returned they are shifted
 *
 * @param sHex
 * @return
 */
public static int[][] aesShiftRow(int[][] sHex) {
  int[][] outStateHex = new int[4][4];
  int counter = 4;
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
      if (i > 0)
        outStateHex[i][(j + counter) % 4] = sHex[i][j];

      else
        outStateHex[i][j] = sHex[i][j];
    }
    counter--;
  }
  return outStateHex;
}



/* *PERF* Replaced the mix columns functions with a lookup table implementation based on my Lab5 */

/**
 * Performs Rjindael column mixing (multiply column with circulant matrix)
 * Galosian multiplication implemented as function, addition is  XOR
 * @param inCol 4 entry byte array representing a column of the state matrix
 * @return  4 entry byte array of mixed column
```

```
   */
  static int[] AESMixColumn(int[] column) {
    int[] result_column = new int[4];
    result_column[0] = GMUL_BY2[column[0]] ^ GMUL_BY3[column[1]] ^          column[2]  ^
column[3];
    result_column[1] =          column[0]  ^ GMUL_BY2[column[1]] ^ GMUL_BY3[column[2]] ^
column[3];
    result_column[2] =          column[0]  ^          column[1]  ^ GMUL_BY2[column[2]] ^
GMUL_BY3[column[3]];
    result_column[3] = GMUL_BY3[column[0]] ^          column[1]  ^          column[2]  ^
GMUL_BY2[column[3]];
    return result_column;
  }

  /**
   * Wrapper around Rjindael column mixing function
   *  Need a way to extract columns as an array for mixing
   * @param inState 4x4 matrix of integer bytes
   * @return 4x4 matrix of integer bytes, shaken and stirred
   */
  static int[][] AESMixColumns(int[][] inState) {
    int[][] result_matrix = new int[4][4];

    for (int column = 0; column < 4; column++) {
      int[] column_array = new int[4];

      for (int row = 0; row < 4; row++) {
        column_array[row] = inState[row][column];
      }

      int[] mixed_column = AESMixColumn(column_array);

      for (int row = 0; row < 4; row++) {
        result_matrix[row][column] = mixed_column[row];
      }
    }

    return result_matrix;
  }

  /* *PERF* Removed GMultiplication functions to replace with lookup tables */
}
```