12-1-2020

A Fortified Extension of the AES and Its Implementation

Ashby Mullin

A FORTIFIED EXTENSION OF THE AES AND ITS IMPLEMENTATION

By

Ashby Mullin

Bachelor of Science – Computer Science
Doane University, Crete
2018

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2020

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

November 20, 2020

This thesis prepared by

Ashby Mullin

entitled

A Fortified Extension of the AES and Its Implementation

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Evangelos Yfantis, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

Andreas Stefik, Ph.D.
Examination Committee Member

John Minor, Ph.D.
Examination Committee Member

Sarah Harris, Ph.D.
Graduate College Faculty Representative

Abstract

With the advancement of quantum computing (QC), the integrity of cryptography has been called into question [1, 2, 3]. For example, two QC algorithms have been developed that can break asymmetric encryption (i.e., Grover's, Shor's), which also poses a threat to symmetric encryption. Asymmetric encryption efforts addressing this threat include lattice-based cryptography, which uses lattice problems to reduce efficiency of cryptanalysis. Symmetric encryption security can be bolstered by increasing the key length, allowing for additional permutations a key could have; known as keyspace. This thesis seeks to expand the keyspace of symmetric encryption in order to create more possibilities. This fortification to the Advanced Encryption Standard (AES) would increase the complexity of encryption and make cipher texts more resistant to attacks. A key component of the AES is the application of byte-substitution using a predetermined mapping. We propose the possibility to change mapping between rounds while still being able to encrypt and decrypt messages without loss of information. The process described in this paper allows the user to choose the substitution box mapping for each round and would increase the keyspace to create 4.97×10^{86} unique permutations compared to 3.4×10^{38} permutations currently afforded by 128-bit AES encryption. By enacting this type of fortification, the AES becomes more robust, protecting data against QC attacks.

Acknowledgments

I dedicate this work to my loving wife & partner Dr. Tina Vo; she has always been there to support me through long nights and difficult problems with encouraging words and thoughtful feedback.

Also, I want to express the utmost appreciation for my advisor, Dr. Yfanits, without his mentorship and guidance this research would not have been possible. His dedication to thouroughly explaining complex ideas has given me greater insight into the fundamentals of the field. I will carry what I have learned from him with me as I continue to grow as a Computer Scientist.

I would like to thank the members of my committee, Dr. Andreas Stefik, Dr. John Minor, and Dr. Sarah Harris, for taking the time to read this thesis and provide suggestions for its improvement. This group of intellectuals have guided my thinking and helped me to define concepts, perform emperical research, and communicate findings and results.

Finally, I would like to give my thanks to Dr. Mark Meysenburg and Dr. Alec Engebretson for spurring my intrests in programming and research which led me to pursue this path.

P.S. My cats Hannibal and Gregory have also been a great treasure in this time. Thank you for encouraging me to nap and sit in the sun.

Table of Contents

List of Figures

List of Algorithms

Chapter 1: Introduction

Currently, cryptography is ubiquitous within our digital communication systems, whether communicating through daily email or transmitting highly sensitive information within local networks. Guaranteeing the security of information, through cryptography, is a growing concern as quantum computer technology evolves and more sophisticated processes and techniques are needed to ensure privatization [4, 5]. Modern threats exist across both asymmetric and symmetric cryptography. Asymmetric cryptography occurs when properties around the factorization of primes are used to create public keys. Multiple types of asymmetric algorithms have been developed [6, 7], promoting public key exchange and digital signature verification. Problematically, Shore's algorithm can compromise this type of encryption by exploiting quantum computing's ability to simulate *states*.

This paper focuses on symmetric cryptography which occurs when both parties have access to identical cipher keys, which guides a series of systematic transformations and inverse transformations to protect data. Currently, only Grovers' algorithm threatens symmetric encryption by reducing the amount of time needed to search keyspace by providing a quadratic speedup over classical computing [8, 9, 10, 11]. Much like asymmetric cryptography, multiple types of symmetric encryption exist including the industry standard, AES [6, 12, 13, 14]. This thesis provides empirical data towards fortifying AES by increasing the keyspace through user-selected S-Boxes to help obfuscate the pathway between ciphertext and plain text, asking the research question:

Can user-selected S-Boxes fortify AES encryption by increasing keyspace?

The following sections (1) provide background information and operationalize important terms foundational to AES, (2) describe and verify the methods used to generate S-Boxes to fortify AES encryption, (3) demonstrate code implementation, and finally (4) supply conclusions and limitations to this research.

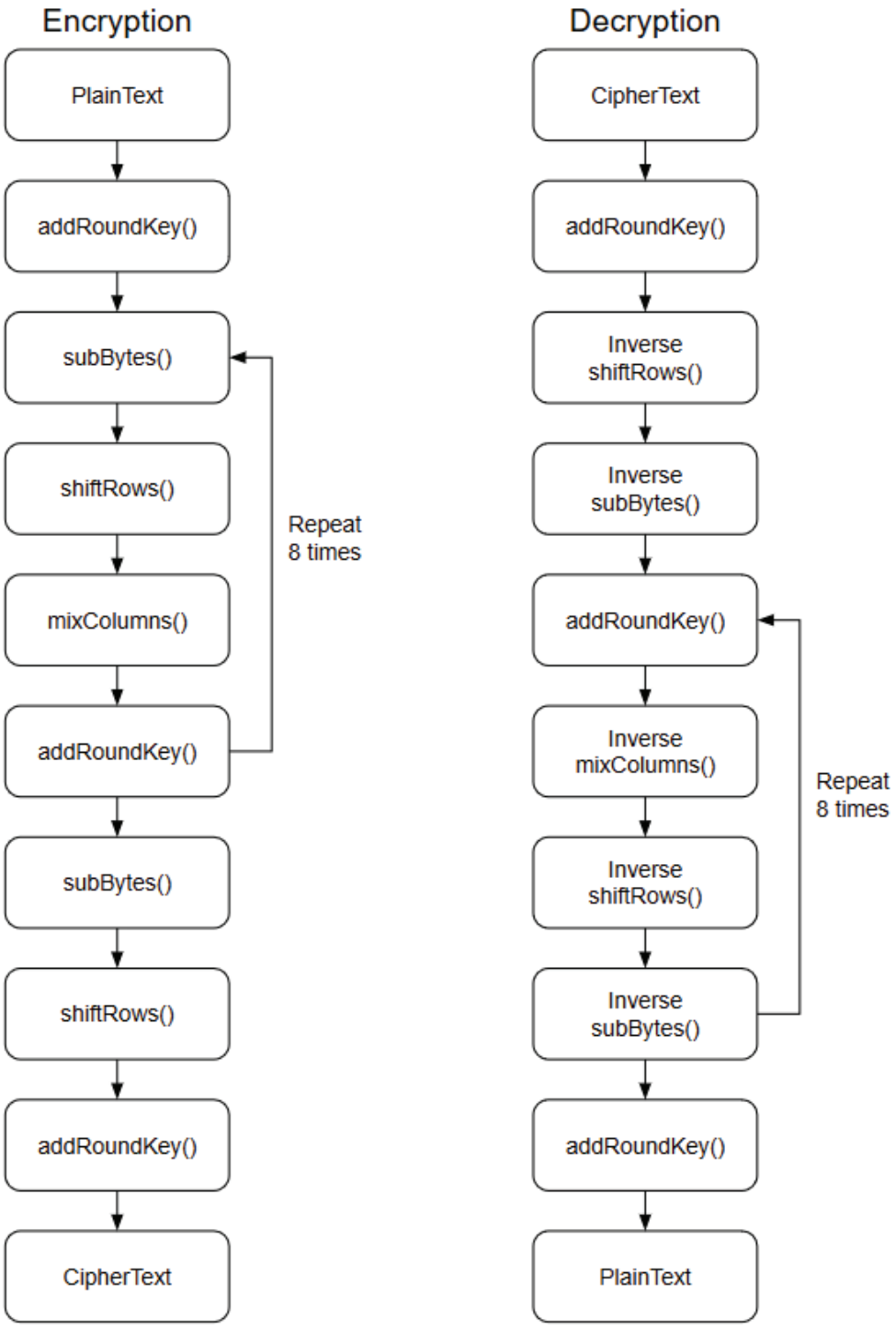
Chapter 2: Background

2.1 The Advanced Encryption Standard

The Advanced Encryption Standard (AES) was introduced in 2001 by the National Institute of Standards and Technology (NIST) after it was determined the Data Encryption Standard was no longer secure [6]. There were 15 algorithms submitted to this call including MARS, RC6, Rijndael, Serpent and Twofish. The Rijndael algorithm was selected by NIST to be the AES [12, 13, 14, 15]. The Rijndael Algorithm is a symmetric block cipher that uses a series of transformations performed in rounds to encode a message, known as plaintext, into a scrambled message that resembles noise, known as ciphertext. Rijndael operates on a message by splitting that message into sections of 128 bits called blocks and uses a sequence of 128, 192, or 256 bits as a cipherkey. There are three types of rounds that can occur during Rijndael encryption and decryption; the initial round, the normal round, and the final round.

In encryption the initial round consists of an addRoundKey() operation. For the normal round the following transformations are performed in order: subBytes(), shiftRows(), mixColumns(), and addRoundKey(). The final round applies the same transformations as the normal round with the exception of mixColumns(). With decryption, all of the transformations must be undone thus all decryption uses an inverse version of each operation in a reverse order to that of encryption. This means that the initial decryption round performs addRoundKey(), invShiftRows, and invSubBytes(). While the normal round consists of addRoundKey(), invMixColumns(), invShiftRows(), and invSubBytes(). The final round performs only an addRoundKey() operation. The encryption and decryption processes are shown in figure 1.

Figure 1: Rijndael Encryption / Decryption



2.1.1 addRoundKey():

The addRoundKey() transformation is where a bitwise XOR operation is performed using the appropriate round key from the key schedule and the current state of the block to produce a new block state. Notice that this operation does not have an inverse version as applying the addRoundKey() using the new state and same round key will return the state prior to the addRoundKey() transformation.

A key schedule is created using the key expansion algorithm, which uses the cipher key to derive round keys in the following manner.

Figure 2: Example of addRoundKey()

	Byte Values				Binary Values			
State:	DF	CA	9F	4D	11011111	11001010	10011111	01001101
Round Key:	B6	03	32	4D	10110110	00000011	00110010	01001101
New State:	69	C9	AD	00	01101001	11001001	10101101	00000000

2.1.2 subBytes() / invSubBytes():

The subBytes() operation is a nonlinear transformation where each byte in a block is replaced by another byte. A substitution byte can be generated at the time of substitution, but this can be computationally expensive. Thus it is more common to see a byte replaced using a pre generated mapping of bytes to substitution bytes. This mapping is called a substitution box (S-Box) and allows for the algorithm to use the byte being replaced to find the substitution byte. To return the original byte during an invSubBytes() operation a inverse S-Box is used that maps the

substitution byte to the original byte and a lookup can be performed using the substitution byte to get the original byte.

Figure 3: AES S-Box and Inverse S-Box

S-Box		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Inverse S-Box		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Given the byte value 7D, look at the left nibble to get x = 7. Next, look at the right nibble to get y = D. Using the S-Box chart shown above we see the value at 7,D is FF and will be used to replace 7D in the block. This replacement is a subByte() transformation. To reverse this as is the case with an invSubByte() look at FF where x = F and y = F and use the Inverse S-Box chart. The value at FF is 7D and would be used to replace FF with the original value of 7D.

2.1.3 shiftRows() / invShiftRows():

Structuring the block as a 4 by 4 array of bytes shown in figure 2, a shiftRows() operation will shift each row cyclically to the left by n_r-1 where n_r is the row number. That is the first row is not shifted at all, the second row is shifted by 1 position and so on. The inverse ShiftRows() undoes the shiftRows() by performing the same cyclic shift but to the right instead of left.

Figure 4: Example of shiftRows()

62	96	B2	1C	No shift	62	96	B2	1C
B5	40	B9	07	Shift left by 1	40	B9	07	B5
FA	B3	D9	5A	Shift left by 2	D9	5A	FA	B3
77	13	02	02	Shift left by 3	02	77	13	02

Figure 5: Example of invShiftRows()

62	96	B2	1C	No shift	62	96	B2	1C
40	B9	07	B5	Shift right by 1	B5	40	B9	07
D9	5A	FA	B3	Shift right by 2	FA	B3	D9	5A
02	77	13	02	Shift right by 3	77	13	02	02

2.1.4 mixColumns() / invMixColumns():

The mixColumns transformation is a matrix multiplication using the block and either the mix columns matrix or inverse mix columns matrix shown in figure 6. This operation provides additional diffusion to block being enciphered.

Figure 6: Matrices used for mixColumns() and invMixColumns()

Mix Columns Matrix

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

Inverse Mix Columns Matrix

0E	0B	0D	09
09	0E	0B	0D
0D	09	0E	0B
0B	0D	09	0E

Figure 7: Example of mixColumns

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

Mix Columns Matrix

X

62	96	B2	1C
40	B9	07	B5
D9	5A	FA	B3
02	77	13	02

Block

=

DF	CA	9F	4D
90	66	BA	A1
8D	02	6F	D2
3B	AC	16	26

New Block

Looking at figure 7 above the matrix multiplication to get the up left value of the product matrix the equation would be 62x02+40x03+D9x01+02x01. However, these bytes are just a base 16 representation of a polynomial. Converting values to their base 2 representations provides the

following equation: 01100010x00000010 + 01000000x00000011 + 11011001x00000001 + 00000010x00000001 and can be thought of as the following polynomial equation $[(x^6 + x^5 + x) * x] + [x^6 * x + 1] + [(x^7 + x^6 + x^4 + x^3 + 1) * 1] + [x * 1]$.

Which is equivalent to $x^7 + x^6 + x^2 + x^7 + x^6 + x^7 + x^6 + x^4 + x^3 + 1 + x$ and simplifies to $3x^7 + 3x^6 + x^4 + x^3 + x^2 + x + 1$. Since this occurs in the $GF(2^8)$ each coefficient is modulo 2 resulting in the equation $x^7 + x^6 + x^4 + x^3 + x^2 + x + 1$ which is represented as the binary string 11011111 and has a byte value of DF.

A similar process occurs during the inverseMixColumns operation by replacing the mix columns matrix with the inverse mix columns matrix and the block with the new block, shown in figure 8. Should the degree of the polynomial reach 8 the polynomial degree should be reduced using a modular base such as $x^8 + x^4 + x^3 + x + 1$ should be applied by performing a bitwise xor operation using the polynomial and the modular base.

Figure 8: Example of invMixColumns

0E	0B	0D	09
09	0E	0B	0D
0D	09	0E	0B
0B	0D	09	0E

X

DF	CA	9F	4D
90	66	BA	A1
8D	02	6F	D2
3B	AC	16	26

=

62	96	B2	1C
40	B9	07	B5
D9	5A	FA	B3
02	77	13	02

2.2 Substitution Boxes

The S-Box and inverse S-Box play a large role not only in the encryption and decryption algorithm but also the key expansion algorithm. S-Boxes are directly responsible for how secure the resulting ciphertext is [16]. For this reason, there have been many studies in the generation

and properties of S-Boxes to determine what constitutes a strong S-Box [17, 18, 19]. Currently, there are two main methods of using S-Boxes. The first is the static S-Box method where an S-Box is chosen prior to any encryption or decryption and is used throughout the process of enciphering and deciphering a message [20, 21, 22]. The other is the dynamic S-Box method where the S-Box itself is changed in some manner and the algorithm progresses through the rounds [23, 24, 25]. The dynamic method is thought to be more secure since there is less of a relationship between the plaintext and the ciphertext, though it is difficult to measure the properties of a dynamic S-Box. The AES uses a predefined static S-Box and its inverse shown below.

While most of the time this S-Box is either hard coded in a piece of software or provided as a hardware implementation, it is possible to programmatically generate this S-Box through the use of the extended euclidean algorithm, and modular algebra.

First it is important to know how the bit pattern of a byte can be imagined as a polynomial in a galois field (GF). A galois field (p^n) indicates a finite field with p^n elements and polynomials with a degree no higher than n-1. Thus GF(2^8) has 256 elements and polynomials with a maximum degree of 7. Considering the byte value 63, the associated bit pattern is 01100011. Let each bit of the pattern be a coefficient of x^n where n is n^{th} position from the right starting at 0. Then, 01100011 can be seen as the polynomial $0x^7 + 1x^6 + 1x^5 + 0x^4 + 0x^3 + 0x^2 + 1x + 1$ or more compactly $x^6 + x^5 + x + 1$. Since each byte has an associated polynomial in GF(2^8) it becomes possible to perform algebraic operations on bytes, such as modular addition, as seen in addRoundKey(), and finding the multiplicative inverse using the extended euclidean algorithm.

Chapter 3: Methodology

The findInv algorithm is an adjustment on the extended euclidean algorithm that returns the multiplicative inverse of a value in the GF(2^8). A few of the adjustments that have been made are operations such as division and multiplication have been adjusted to use the 8-bit representation of a byte value so that polynomial multiplication and division can be achieved as shown in the polyDiv and polyMult algorithms. Additionally, checking for non-invertibility has been removed since all operations are in the GF(2^8) which is a finite field meaning that all elements will have an inverse element.

Algorithm 1: findInv

```
 1  findInv(aByte, mod)
 2      if(aByte == 1 or aByte == 1)
 3          return aByte
 4      initQR = divide(mod, aByte)
 5      invArr = new array[3][3]
 6      invArr[0] = [mod, 1, 0]
 7      invArr[1] = [aByte, 0, 1]
 8      idx = 2
 8      while(true)
 9        qr = divide(invArr[(idx-2)%3][0], invArr[(idx-1)%3)][0])
10        invArr[idx%3][0] = qr[1]
11        invArr[idx%3][1] = invArr[(idx-2)%3][1] ^
                             polyMult(invArr[(idx-1)%3][1],qr[0])
12        invArr[idx%3][2] = invArr[(idx-2)%3][2] ^
                             polyMult(invArr[(idx-1)%3][2],qr[0])
13        if(invArr[idx%3][0] == 1)
14            return invArr[idx%3][2]
15        idx++
```

Algorithm 2: divide

```
 1  divide(dividend, divisor)
 2      quotient = 0
 3      remainder = dividend
 4      resultSize = degree of dividend
 5      divisorSize = degree of divisor
 6      while(resultSize >= divsiorSize)
 7          tempQuotient = 1 << (resultSize - divisorSize)
 8          quotient = quotient ^ tempQuotient
 9          tempDivisor = divisor << (resultSize - divisorSize)
10          remainder = remainder ^ tempDivisor
11          resultSize = bitCount(remainder)
12      return [quotient, remainder]
```

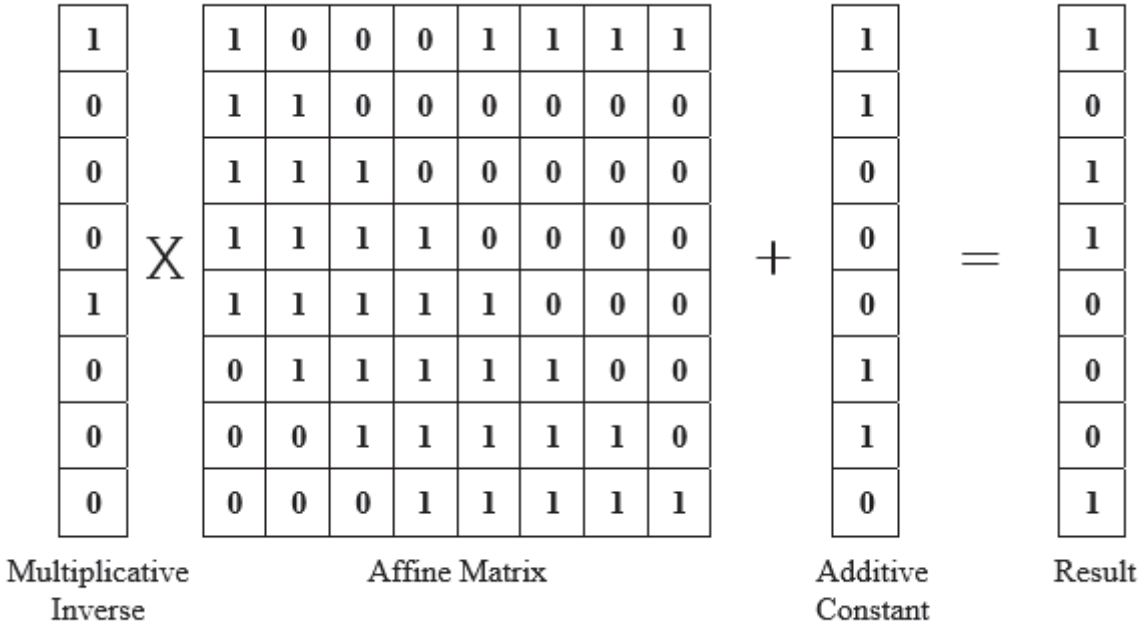
Algorithm 3: polyMult

```
 1  polyMult(aByte, bByte)
 2      product = 0
 3      aBits = 8 bit representation of aByte
 4      bBits = 8 bit representation of bByte
 5      for 0 <= i < 8
 6          if(aBits[i] == 1)
 7              aVal = 1 << (7 - i)
 8              aTemp = 0;
 9              for 0 <= j < 8
10                  if(bBits[j] == 1)
11                      bTemp = aVal << (7 - j)
12                      aTemp = aTemp ^ bTemp
13              product = product ^ aTemp
14      return product
```

These algorithms supply the appropriate multiplicative inverse that is used in the affine transformation which gives the substitution byte for a given byte value in the GF(2^8) with a particular modulus. This multiplicative inverse is needed for the affine transformation that provides the substitution byte for a byte value. Consider the byte B4, using the findInv algorithm gives the multiplicative inverse byte 11 which is 00010001 in binary. Appling the affine

transformation shown below results in the binary string 10001101 or 8D which is the substitution byte for B4.

Figure 9: Affine transformation generating a substitution byte



The genSubByte algorithm below performs this affine transformation given a byte value, an affine matrix, additive constant, and an indication if the byte value being passed is the substitution byte. This allows for the algorithm to generate both a substitution byte for given byte value and vice versa.

Algorithm 4: genSubByte

```
 1  genSubByte(aByte, mod, aMatrix, aVector, invert)
 2      invVector = new array[8]
 3      if invert
 4          invVector = byteToVec(aByte)
 5      else
 6          inverse = findInv(aByte, mod)
 7          invVector = byteToVec(inverse)
 8      invVector = matVecMul(aMatrix, invVector)
 9      invVector = vecAdd(inVector, aVector)
10      subVal = vecToByte(invVector)
11      if invert
12          return findInv(subVal, mod)
11      return subVal
```

Algorithm 5: sBoxGen

```
 1  sBoxGen(mod, invert)
 2      aMatrix = new array[8][8]
 3      aVector = new array[8]
 4      if invert
 5          aMatrix = invMatrix
 6          aVector = invOffsetVector
 7      else
 8          aMatrix = baseMatrix
 9          aVector = baseOffsetVector
10      sBox = new array[256]
11      for 0 <= i <= 255
12          sBox[i] = genSubByte(i, mod, aMatrix, aVector, invert)
13      return sBox
```

With the algorithms provided up to this point it is now possible to generate an S-Box and its associated inverse using any of the irreducible polynomials in $GF(2^8)$. For instance if the mod passed to sBoxGen is $x^8+x^4+x^3+x+1$ then the AES S-Box and its inverse can be generated. However, if a different polynomial is chosen as the modulus base such as $x^8+x^6+x^5+x^4+x^2+x+1$ is chosen then a different pair of S-Boxes are generated.

Figure 10: S-Boxes generated using $x^8+x^6+x^5+x^4+x^2+x+1$ as the modulo base

S-Box		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	63	7C	27	E4	8A	1E	A0	DE	97	08	DD	BE	49	07	76	B7
	1	D2	D0	1D	E6	F7	0F	8D	CD	BD	8C	51	C8	22	28	09	3D
	2	BB	DB	BA	C2	5C	11	A1	A5	29	FA	55	B4	DF	77	FF	52
	3	0C	45	5F	7F	B1	6D	B6	E1	C3	8B	C6	69	9D	57	87	47
	4	C4	80	3F	D6	44	AA	B3	02	FC	0E	91	3B	C9	9A	00	95
	5	46	AB	64	F3	78	E5	88	84	F6	E2	A2	75	2D	C5	30	14
	6	D4	98	70	F4	7D	8E	A6	61	C1	21	AF	15	89	94	E9	B2
	7	33	FE	17	9B	7A	23	66	6F	D7	54	79	EB	DA	D1	71	5A
	8	7B	74	92	CF	86	06	72	0D	F0	2F	4C	03	C0	1B	18	E7
	9	AC	F9	D5	65	1A	B9	4F	60	36	67	9F	A4	19	90	D3	2E
	A	F1	E8	CC	B8	2B	9C	E0	25	EE	10	20	85	96	DC	5B	C7
	B	A9	2C	68	04	48	F5	A3	BC	8F	99	FB	5D	CA	3C	D8	B5
	C	73	83	9E	B0	EA	82	A8	D9	A7	31	5E	39	81	0A	62	24
	D	32	CE	42	38	05	7E	58	CB	16	37	53	35	ED	26	40	50
	E	4B	4E	AD	FD	59	4A	1F	01	EF	0B	43	E3	2A	1C	AE	12
	F	F2	41	F8	56	6E	13	EC	6B	BF	3E	3A	4D	6A	93	34	6C

Inverse S-Box		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	4E	E7	47	8B	B3	D4	85	0D	09	1E	CD	E9	30	87	49	15
	1	A9	25	EF	F5	5F	6B	D8	72	8E	9C	94	8D	ED	12	05	E6
	2	AA	69	1C	75	CF	A7	DD	02	1D	28	EC	A4	B1	5C	9F	89
	3	5E	C9	D0	70	FE	DB	98	D9	D3	CB	FA	4B	BD	1F	F9	42
	4	DE	F1	D2	EA	44	31	50	3F	B4	0C	E5	E0	8A	FB	E1	96
	5	DF	1A	2F	DA	79	2A	F3	3D	D6	E4	7F	AE	24	BB	CA	32
	6	97	67	CE	00	52	93	76	99	B2	3B	FC	F7	FF	35	F4	77
	7	62	7E	86	C0	81	5B	0E	2D	54	7A	74	80	01	64	D5	33
	8	41	CC	C5	C1	57	AB	84	3E	56	6C	04	39	19	16	65	B8
	9	9D	4A	82	FD	6D	4F	AC	08	61	B9	4D	73	A5	3C	C2	9A
	A	06	26	5A	B6	9B	27	66	C8	C6	B0	45	51	90	E2	EE	6A
	B	C3	34	6F	46	2B	BF	36	0F	A3	95	22	20	B7	18	0B	F8
	C	8C	68	23	38	40	5D	3A	AF	1B	4C	BC	D7	A2	17	D1	83
	D	11	7D	10	9E	60	92	43	78	BE	C7	7C	21	AD	0A	07	2C
	E	A6	37	59	EB	03	55	13	8F	A1	6E	C4	7B	F6	DC	A8	E8
	F	88	A0	F0	53	63	B5	58	14	F2	91	29	BA	48	E3	71	2E

Notice that this pair of S-Boxes are, except for two entries (00 and 01), are entirely different from the AES S-Boxes. This suggests each irreducible polynomial in $GF(2^8)$ can be used to generate a significantly different S-Box to any other S-Box generated by a different irreducible polynomial. However, a useful S-Box will have only a single mapping to each value between 0 and 255. To check this the verify algorithm creates an empty set to check if has

already been seen as the algorithm traverses the entire mapping. If a value is not in the set it has not been seen yet and then is added to the set. If a value is in the set then it has been seen before and the S-Box contains duplicate values so it is not useful. The inverse S-Box is also checked in this manner for thoroughness. While traversing the S-Box, this algorithm also checks that the inverse S-Box provides a mapping back to the original byte ensuring that a byte substitution can be undone. In this way the algorithm confirms the mapping between an S-Box and its inverse is bijective.

Algorithm 6: sBoxVerify

```
 1  sBoxVerify(sBox, isBox)
 2      curBase = new set
 3      curInv = new set
 4          for 0 <= i < 256
 5              if sBox[i] not in curBase
 6                  add sBoxes[i] to curBase
 7              else
 8                  there is a duplicate
 9              if isBox[i] not in curInv
10                  add isBox[i] to curInv
11              else
12                  there is a duplicate
13              iByte = subByte(i, sBox)
14              sByte = subByte(iByte, isBox)
15              if iByte sByte != i
16                  the sBox is not bijective
```

Not only should an S-Box contain one of each value in the range of 0 to 255 and bijective with respect to an associated inverse S-Box. The S-Boxes itself should be unique thus algorithm 7 was developed to compare how similar any two S-Boxes generated using algorithm 5 are by checking the position of byte values within S-Boxes.

Algorithm 7: sBoxCompare

```
 1  sBoxCompare()
 2      mods = array containing irreducible polynomials in GF(2^8)
 3      sBoxes = new array[30][255]
 4      isBoxes = new array[30][255]
 5      for 0 <= i < 30
 6          sBoxes[i] = sBoxGeneration(mods[i], true)
 7          isBoxes[i] = sBoxGeneration(mods[i], false)
 8      simMatrix = new array[30][30]
 9      for 0 <= i < 30
10          for 0 <= j < 256
11              for 0 <= k+i < 30
12                  if sBoxes[i][j] == sBoxes[k+i][j]
13                      simMatrix[i][k+i] += 1
14          for 0 <= j <= 30
15              simMatrix[i][j] /= 256
16      return simMatrix
```

Algorithm 8: fortKeyExpansion

```
 1  fortKeyExpansion(cipherKey,sequence,sBoxes)
 2      rcon = buildRcon(cipherKey)
 3      words = new array[44][4]
 4      for 0 <= i < 4
 5              for 0 <= j < 4
 6                   words[i][j] = cipherKey[4*i+j]
 7         for 4 <= i < 44
 8              word = new array[4]
 9              for 0 <= j < 4
10                   word[j] = words[i-1][j]
11              if i mod 4 is 0
12                   word = subWord(rotWord(word), sBoxes(i/4))
13                   word[0] xor rcon[(i/4)-1]
14              for 0 <= j < 4
15                   words[i][j] =  word[j] xor words[i-4][j]
16         roundKeys = new array[11][16]
17         for 0 <= i < 11
18              for 0 <= j < 16
19                   roundKeys[i][j] = words[j/4+i*4][j%4]
20         return roundKeys
```

In algorithm the standard Key Expansion algorithm is modified to accept a sequence as a parameter. This sequence consists of integer values indicating the index of a particular S-Box to be used from the set of S-Boxes that are passed in using the parameter sBoxes. The indecies of sequence are designed to correlate with the rounds (i.e. the initial round will look at sequence[0] to determine the S-Box to be used, where round 1 will look at sequence[1]). The subByte Algortihm performs the actual byte replacement by peforming the look up using the passed in byte value and S-box to return the appropiate substitution values. Incidentally, there is no longer a need for a invSubByte() algorithm as the subByte algorithm can accomplish the task when passed a substiturion value and inverse S-Box.

Algorithm 9: subWord

```
1  subWord(word, sBox)
2      for byte in word
3          byte = subByte(oByte, sBox)
4      return word
```

Algorithm 10: subByte

```
1  subByte(aByte, sBox)
2      lookup = {aByte >> 4, aByte ^ (left << 4)}
3      return sBox[lookup[0] * 16 + lookup[1]]
```

This same concept is applied to the Rijndael algorithm when performing encryption and decryption. Note that the subBytes() operation is functionally equivalent to the subWord() algorithm the only difference is in implementation where subWords() operates a word containing 4 bytes, subBytes operates on a block of 16 bytes.

Algorithm 11: fortAesEncryption

```
 1  fortAesEncryption(plainText,cipherKey,sequence,sBoxes)
 2      blocks = splitBlocks(plainText)
 3      roundKeys = keyExpansion(cipherKey,sequence,sBoxes)
 4      for block in blocks
 5          block = addRndKey(block,roundKeys[0])
 6          for 1 < = i < 10
 7              block = transBlock(block)
 8              block = subBytes(block,sBoxes[sequence[i]])
 9              block = shiftRows(block)
10              block = mixColumn(block)
11              block = addRndKey(block,roundKeys[i])
12          block = subBytes(block,sBoxes[sequence[10]])
13          block = shiftRows(block)
14          block = addRndKey(block,roundKeys[10])
15      cipherText = combineBlocks(blocks)
16      return cipherText
```

Algorithm 12: fortAesDecryption

```
 1  fortAesDecryption(plainText, cipherKey, sequence, isBoxes)
 2      blocks = splitBlocks(plainText)
 3      roundKeys = keyExpansion(cipherKey, sequence, isBoxes)
 4      for block in blocks
 5          block = addRndKey(block, roundKeys[10])
 6          block = shiftRows(block)
 7          block = subBlock(block, isBoxes[sequence[10]])
 8          for 9 < = i < 10
 9              block = addRndKey(block, roundKeys[i])
10              block = invMixColumn(block)
11              block = invShiftRows(block)
12              block = subBlock(block, isBoxes[sequence[i]])
13          block = addRndKey(block, roundKeys[0])
14      block = transBlock(block)
15      plainText = combineBlocks(blocks)
16      return plainText
```

Algorithms 13 through 15 are helper functions that provide Fortified AES the ability to manipulate the message being sent whether it starts as a plain text or cipher text. These algroithms enable the spliting of a message into blocks to be encrypted or decrypted, the

transposale of blocks so that non linearity can be achieved and tranformations applied correctly, and the ability to recombine blocks into a cipher text (message).

Algorithm 13: splitBlocks

```
splitBlocks(plainText)
    nBlocks = length of plainText bytes divide by 16 + 1
    blocks = new array [nBlks][16]
    for 0 <= i < nBlocks
        for 0 <= j < 16
            blocks[i][j] = plainText[i * 16 + j]
    return blocks
```

Algorithm 14: transposeBlock

```
transposeBlock(block)
    tBlock = new array[16]
    for 0 <= i < 16
        tBlock[i] = block[(i/4) + 4 * (i%4)]
    return tBlock
```

Algorithm 15: combineBlocks

```
combineBlocks(blocks)
    msgSize = number of rows blocks has multiplied by 16
    message = new array[msgSize]
    for 0 <= i < msgSize
        for 0 <= j < 16
             message[i/16 + j] = blocks[i][j]
    return message
```

The algorithms discussed in this chapter form the basis of the Fortified AES, a modification to the standard AES that allows a user to define sets of S-Boxes (with their associated inverses) and the order / frequency in which they are used through out the rounds of both the Key Expansion and Rijndael alogrithms.

Chapter 4: Code Demonstration

This program makes use of the algorithms described above to visualize how changing the S-Box used during a particular round in either the key expansion or Rijndael Algorithm. The input and cipher key shown were chosen because they were used in Appendix B of FIPS 197 showing the full transformation from plaintext to ciphertext using the AES S-Box. This was particularly useful to verify the algorithms were performing all transformations correctly throughout the encryption and decryption process. Notice that each round has a drop down menu, these menus allow the user to select a number between 1 and 30 which corresponds to the following list of irreducible polynomials in GF(2^8).

Figure 11: Fortified AES Software Implementation

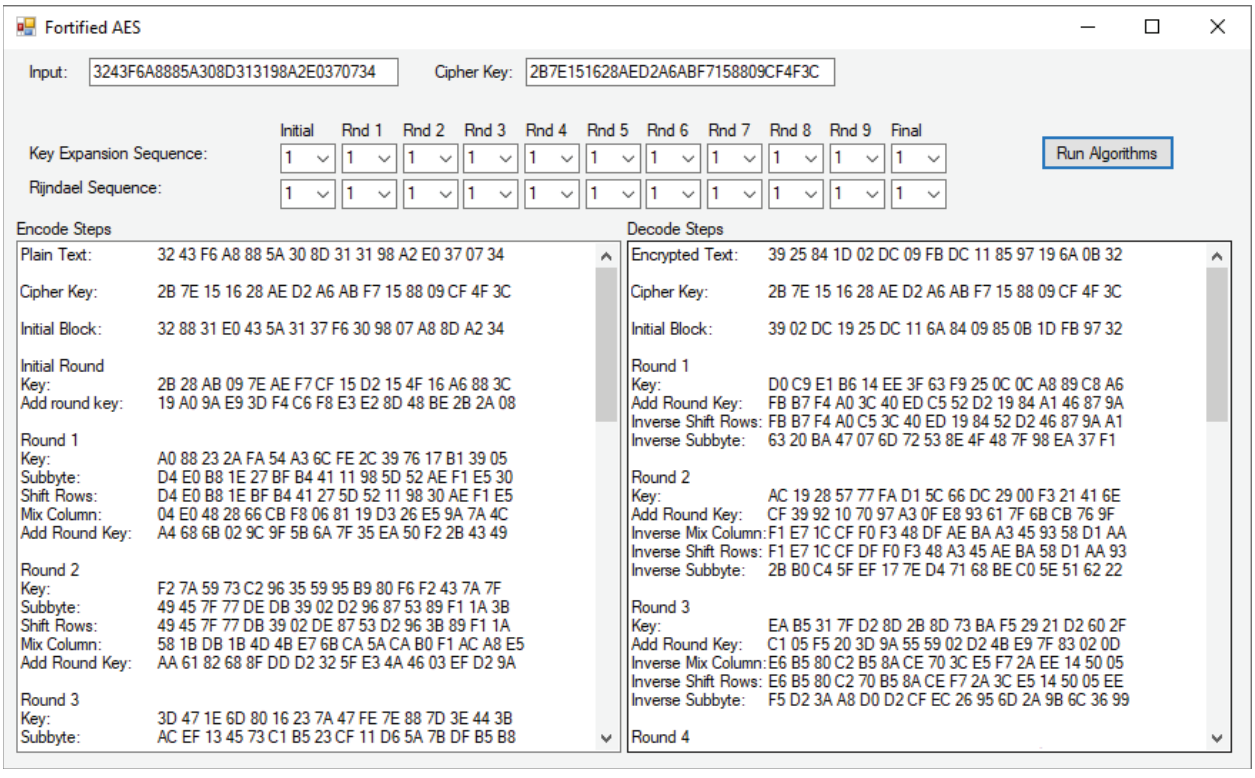


Figure 12: List of Irreducible Polynomials with degree of 8

Polynomial	Decimal Value	Polynomial	Decimal Value
1. $x^8+x^4+x^3+x+1$	283	16. $x^8+x^7+x^3+x+1$	395
2. $x^8+x^4+x^3+x^2+1^*$	285	17. $x^8+x^7+x^3+x^2+1^*$	397
3. $x^8+x^5+x^3+x+1^*$	299	18. $x^8+x^7+x^4+x^3+x^2+x+1$	415
4. $x^8+x^5+x^3+x^2+1^*$	301	19. $x^8+x^7+x^5+x+1$	419
5. $x^8+x^5+x^4+x^3+x+1$	313	20. $x^8+x^7+x^5+x^3+1^*$	425
6. $x^8+x^5+x^4+x^3+x^2+x+1$	319	21. $x^8+x^7+x^5+x^4+1$	433
7. $x^8+x^6+x^3+x^2+1^*$	333	22. $x^8+x^7+x^5+x^4+x^3+x^2+1$	445
8. $x^8+x^6+x^4+x^3+x^2+x+1^*$	351	23. $x^8+x^7+x^6+x+1^*$	451
9. $x^8+x^6+x^5+x+1^*$	355	24. $x^8+x^7+x^6+x^3+x^2+x+1^*$	463
10. $x^8+x^6+x^5+x^2+1^*$	357	25. $x^8+x^7+x^6+x^4+x^2+x+1$	471
11. $x^8+x^6+x^5+x^3+1^*$	361	26. $x^8+x^7+x^6+x^4+x^3+x^2+1$	477
12. $x^8+x^6+x^5+x^4+1^*$	369	27. $x^8+x^7+x^6+x^5+x^2+x+1^*$	487
13. $x^8+x^6+x^5+x^4+x^2+x+1$	375	28. $x^8+x^7+x^6+x^5+x^4+x+1$	499
14. $x^8+x^6+x^5+x^4+x^3+x+1$	379	29. $x^8+x^7+x^6+x^5+x^4+x^2+1^*$	501
15. $x^8+x^7+x^2+x+1^*$	391	30. $x^8+x^7+x^6+x^5+x^4+x^3+1$	505

* indicates a primitive polynomial

The plain text and cipher keys chosen for the following examples where chosen because they appear in FIPS 197 [13], which provides the state of the plain text block after each transformation performed by the AES, creating a method for comparision. Looking through the results in section 4.1 notice that all the transformations are the same as those in FIPS showing the algorithms in this paper are perfoming AES transformations correctly.

4.1 Example of AES Encryption using Fortified AES

Using the S-Box generated by the first irreducible polynomial of degree 8 in each round for both the Key Expansion and Rijndael algorithms, the Fortified AES performs in the exact same manner as the standard AES, providing the same cipher text expected from standard AES.

Figure 13: Sequence of S-Boxes and results for standard AES

	Round										
Algorithm	I	1	2	3	4	5	6	7	8	9	F
Key Expansion	1	1	1	1	1	1	1	1	1	1	1
Rijndael Algorithm	1	1	1	1	1	1	1	1	1	1	1

Plain Text:	32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
Cipher Key:	2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
Initial Block:	32 88 31 E0 43 5A 31 37 F6 30 98 07 A8 8D A2 34
Initial Round Key:	2B 28 AB 09 7E AE F7 CF 15 D2 15 4F 16 A6 88 3C
Add round key:	19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08
Round 1 Key:	A0 88 23 2A FA 54 A3 6C FE 2C 39 76 17 B1 39 05
Subbyte:	D4 E0 B8 1E 27 BF B4 41 11 98 5D 52 AE F1 E5 30
Shift Rows:	D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5
Mix Column:	04 E0 48 28 66 CB F8 06 81 19 D3 26 E5 9A 7A 4C
Add Round Key:	A4 68 6B 02 9C 9F 5B 6A 7F 35 EA 50 F2 2B 43 49
Round 2 Key:	F2 7A 59 73 C2 96 35 59 95 B9 80 F6 F2 43 7A 7F
Subbyte:	49 45 7F 77 DE DB 39 02 D2 96 87 53 89 F1 1A 3B
Shift Rows:	49 45 7F 77 DB 39 02 DE 87 53 D2 96 3B 89 F1 1A
Mix Column:	58 1B DB 1B 4D 4B E7 6B CA 5A CA B0 F1 AC A8 E5
Add Round Key:	AA 61 82 68 8F DD D2 32 5F E3 4A 46 03 EF D2 9A
Round 3 Key:	3D 47 1E 6D 80 16 23 7A 47 FE 7E 88 7D 3E 44 3B

```
Subbyte:            AC EF 13 45 73 C1 B5 23 CF 11 D6 5A 7B DF B5 B8
Shift Rows:         AC EF 13 45 C1 B5 23 73 D6 5A CF 11 B8 7B DF B5
Mix Column:         75 20 53 BB EC 0B C0 25 09 63 CF D0 93 33 7C DC
Add Round Key:      48 67 4D D6 6C 1D E3 5F 4E 9D B1 58 EE 0D 38 E7

Round 4
Key:                EF A8 B6 DB 44 52 71 0B A5 5B 25 AD 41 7F 3B 00
Subbyte:            52 85 E3 F6 50 A4 11 CF 2F 5E C8 6A 28 D7 07 94
Shift Rows:         52 85 E3 F6 A4 11 CF 50 C8 6A 2F 5E 94 28 D7 07
Mix Column:         0F 60 6F 5E D6 31 C0 B3 DA 38 10 13 A9 BF 6B 01
Add Round Key:      E0 C8 D9 85 92 63 B1 B8 7F 63 35 BE E8 C0 50 01

Round 5
Key:                D4 7C CA 11 D1 83 F2 F9 C6 9D B8 15 F8 87 BC BC
Subbyte:            E1 E8 35 97 4F FB C8 6C D2 FB 96 AE 9B BA 53 7C
Shift Rows:         E1 E8 35 97 FB C8 6C 4F 96 AE D2 FB 7C 9B BA 53
Mix Column:         25 BD B6 4C D1 11 3A 4C A9 D1 33 C0 AD 68 8E B0
Add Round Key:      F1 C1 7C 5D 00 92 C8 B5 6F 4C 8B D5 55 EF 32 0C

Round 6
Key:                6D 11 DB CA 88 0B F9 00 A3 3E 86 93 7A FD 41 FD
Subbyte:            A1 78 10 4C 63 4F E8 D5 A8 29 3D 03 FC DF 23 FE
Shift Rows:         A1 78 10 4C 4F E8 D5 63 3D 03 A8 29 FE FC DF 23
Mix Column:         4B 2C 33 37 86 4A 9D D2 8D 89 F4 18 6D 80 E8 D8
Add Round Key:      26 3D E8 FD 0E 41 64 D2 2E B7 72 8B 17 7D A9 25

Round 7
Key:                4E 5F 84 4E 54 5F A6 A6 F7 C9 4F DC 0E F3 B2 4F
Subbyte:            F7 27 9B 54 AB 83 43 B5 31 A9 40 3D F0 FF D3 3F
Shift Rows:         F7 27 9B 54 83 43 B5 AB 40 3D 31 A9 3F F0 FF D3
Mix Column:         14 46 27 34 15 16 46 2A B5 15 56 D8 BF EC D7 43
Add Round Key:      5A 19 A3 7A 41 49 E0 8C 42 DC 19 04 B1 1F 65 0C

Round 8
Key:                EA B5 31 7F D2 8D 2B 8D 73 BA F5 29 21 D2 60 2F
Subbyte:            BE D4 0A DA 83 3B E1 64 2C 86 D4 F2 C8 C0 4D FE
Shift Rows:         BE D4 0A DA 3B E1 64 83 D4 F2 2C 86 FE C8 C0 4D
Mix Column:         00 B1 54 FA 51 C8 76 1B 2F 89 6D 99 D1 FF CD EA
Add Round Key:      EA 04 65 85 83 45 5D 96 5C 33 98 B0 F0 2D AD C5

Round 9
Key:                AC 19 28 57 77 FA D1 5C 66 DC 29 00 F3 21 41 6E
Subbyte:            87 F2 4D 97 EC 6E 4C 90 4A C3 46 E7 8C D8 95 A6
Shift Rows:         87 F2 4D 97 6E 4C 90 EC 46 E7 4A C3 A6 8C D8 95
Mix Column:         47 40 A3 4C 37 D4 70 9F 94 E4 3A 42 ED A5 A6 BC
```

```
Add Round Key:        EB 59 8B 1B 40 2E A1 C3 F2 38 13 42 1E 84 E7 D2

Round 10
Key:                  D0 C9 E1 B6 14 EE 3F 63 F9 25 0C 0C A8 89 C8 A6
Subbyte:              E9 CB 3D AF 09 31 32 2E 89 07 7D 2C 72 5F 94 B5
Shift Rows:           E9 CB 3D AF 31 32 2E 09 7D 2C 89 07 B5 72 5F 94
Add Round Key:        39 02 DC 19 25 DC 11 6A 84 09 85 0B 1D FB 97 32

Cipher Text:          39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
```

4.2 Example of S-Box substitution in Key Expansion

When changing the S-Box used in the 8th round of key expansion the transformation of bytes is adjusted in the following manner, providing a different cipher text after all transformations.

Figure 14: Sequence and results for S-Box substituion in Key Expansion

	Round										
Algorithm	I	1	2	3	4	5	6	7	8	9	F
Key Expansion	1	1	1	1	1	1	1	1	30	1	1
Rijndael Algorithm	1	1	1	1	1	1	1	1	1	1	1

```
Plain Text:           32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34

Cipher Key:           2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

Initial Block:        32 88 31 E0 43 5A 31 37 F6 30 98 07 A8 8D A2 34

Initial Round
Key:                  2B 28 AB 09 7E AE F7 CF 15 D2 15 4F 16 A6 88 3C
Add round key:        19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08

Round 1
Key:                  A0 88 23 2A FA 54 A3 6C FE 2C 39 76 17 B1 39 05
Subbyte:              D4 E0 B8 1E 27 BF B4 41 11 98 5D 52 AE F1 E5 30
Shift Rows:           D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5
Mix Column:           04 E0 48 28 66 CB F8 06 81 19 D3 26 E5 9A 7A 4C
```

```
Add Round Key:      A4 68 6B 02 9C 9F 5B 6A 7F 35 EA 50 F2 2B 43 49


Round 2
Key:                F2 7A 59 73 C2 96 35 59 95 B9 80 F6 F2 43 7A 7F
Subbyte:            49 45 7F 77 DE DB 39 02 D2 96 87 53 89 F1 1A 3B
Shift Rows:         49 45 7F 77 DB 39 02 DE 87 53 D2 96 3B 89 F1 1A
Mix Column:         58 1B DB 1B 4D 4B E7 6B CA 5A CA B0 F1 AC A8 E5
Add Round Key:      AA 61 82 68 8F DD D2 32 5F E3 4A 46 03 EF D2 9A

Round 3
Key:                3D 47 1E 6D 80 16 23 7A 47 FE 7E 88 7D 3E 44 3B
Subbyte:            AC EF 13 45 73 C1 B5 23 CF 11 D6 5A 7B DF B5 B8
Shift Rows:         AC EF 13 45 C1 B5 23 73 D6 5A CF 11 B8 7B DF B5
Mix Column:         75 20 53 BB EC 0B C0 25 09 63 CF D0 93 33 7C DC
Add Round Key:      48 67 4D D6 6C 1D E3 5F 4E 9D B1 58 EE 0D 38 E7

Round 4
Key:                EF A8 B6 DB 44 52 71 0B A5 5B 25 AD 41 7F 3B 00
Subbyte:            52 85 E3 F6 50 A4 11 CF 2F 5E C8 6A 28 D7 07 94
Shift Rows:         52 85 E3 F6 A4 11 CF 50 C8 6A 2F 5E 94 28 D7 07
Mix Column:         0F 60 6F 5E D6 31 C0 B3 DA 38 10 13 A9 BF 6B 01
Add Round Key:      E0 C8 D9 85 92 63 B1 B8 7F 63 35 BE E8 C0 50 01

Round 5
Key:                D4 7C CA 11 D1 83 F2 F9 C6 9D B8 15 F8 87 BC BC
Subbyte:            E1 E8 35 97 4F FB C8 6C D2 FB 96 AE 9B BA 53 7C
Shift Rows:         E1 E8 35 97 FB C8 6C 4F 96 AE D2 FB 7C 9B BA 53
Mix Column:         25 BD B6 4C D1 11 3A 4C A9 D1 33 C0 AD 68 8E B0
Add Round Key:      F1 C1 7C 5D 00 92 C8 B5 6F 4C 8B D5 55 EF 32 0C

Round 6
Key:                6D 11 DB CA 88 0B F9 00 A3 3E 86 93 7A FD 41 FD
Subbyte:            A1 78 10 4C 63 4F E8 D5 A8 29 3D 03 FC DF 23 FE
Shift Rows:         A1 78 10 4C 4F E8 D5 63 3D 03 A8 29 FE FC DF 23
Mix Column:         4B 2C 33 37 86 4A 9D D2 8D 89 F4 18 6D 80 E8 D8
Add Round Key:      26 3D E8 FD 0E 41 64 D2 2E B7 72 8B 17 7D A9 25

Round 7
Key:                4E 5F 84 4E 54 5F A6 A6 F7 C9 4F DC 0E F3 B2 4F
Subbyte:            F7 27 9B 54 AB 83 43 B5 31 A9 40 3D F0 FF D3 3F
Shift Rows:         F7 27 9B 54 83 43 B5 AB 40 3D 31 A9 3F F0 FF D3
Mix Column:         14 46 27 34 15 16 46 2A B5 15 56 D8 BF EC D7 43
Add Round Key:      5A 19 A3 7A 41 49 E0 8C 42 DC 19 04 B1 1F 65 0C
```

```
Round 8
Key:                4F 10 94 DA 06 59 FF 59 5C 95 DA 06 09 FA 48 07
Subbyte:            BE D4 0A DA 83 3B E1 64 2C 86 D4 F2 C8 C0 4D FE
Shift Rows:         BE D4 0A DA 3B E1 64 83 D4 F2 2C 86 FE C8 C0 4D
Mix Column:         00 B1 54 FA 51 C8 76 1B 2F 89 6D 99 D1 FF CD EA
Add Round Key:      4F A1 C0 20 57 91 89 42 73 1C B7 9F D8 05 85 ED

Round 9
Key:                9F 8F 1B C1 69 30 CF 96 99 0C D6 D0 5E A4 EC EB
Subbyte:            84 32 BA B7 5B 81 A7 2C 8F 9C A9 DB 61 6B 97 55
Shift Rows:         84 32 BA B7 81 A7 2C 5B A9 DB 8F 9C 55 61 6B 97
Mix Column:         77 2C FF 93 28 70 03 29 B3 9B 2E 6D 15 E8 A0 30
Add Round Key:      E8 A3 E4 52 41 40 CC BF 2A 97 F8 BD 4B 4C 4C DB

Round 10
Key:                39 B6 AD 6C 19 29 E6 70 70 7C AA 7A 26 82 6E 85
Subbyte:            9B 0A 69 00 83 09 4B 08 E5 88 41 7A B3 29 29 B9
Shift Rows:         9B 0A 69 00 09 4B 08 83 41 7A E5 88 B9 B3 29 29
Add Round Key:      A2 BC C4 6C 10 62 EE F3 31 06 4F F2 9F 31 47 AC

Cipher Text:        A2 10 31 9F BC 62 06 31 C4 EE 4F 47 6C F3 F2 AC
```

4.3 S-Box substitution in Rijndael Algorithm

By changing the S-Box used in the 5th round of Rijndael Algorithm the transformation of bytes is adjusted so that ending cipher text is different from both the standard AES and the cipher text resulting from the transformations seen in section 4.2.

Figure 15: Sequence and results for S-Box substituion in Rijnael Algorithm

	Round										
Algorithm	I	1	2	3	4	5	6	7	8	9	F
Key Expansion	1	1	1	1	1	1	1	1	1	1	1
Rijndael Algorithm	1	1	1	1	1	23	1	1	1	1	1

Plain Text:	32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
Cipher Key:	2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
Initial Block:	32 88 31 E0 43 5A 31 37 F6 30 98 07 A8 8D A2 34
Initial Round Key:	2B 28 AB 09 7E AE F7 CF 15 D2 15 4F 16 A6 88 3C
Add round key:	19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08
Round 1	
Key:	A0 88 23 2A FA 54 A3 6C FE 2C 39 76 17 B1 39 05
Subbyte:	D4 E0 B8 1E 27 BF B4 41 11 98 5D 52 AE F1 E5 30
Shift Rows:	D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5
Mix Column:	04 E0 48 28 66 CB F8 06 81 19 D3 26 E5 9A 7A 4C
Add Round Key:	A4 68 6B 02 9C 9F 5B 6A 7F 35 EA 50 F2 2B 43 49
Round 2	
Key:	F2 7A 59 73 C2 96 35 59 95 B9 80 F6 F2 43 7A 7F
Subbyte:	49 45 7F 77 DE DB 39 02 D2 96 87 53 89 F1 1A 3B
Shift Rows:	49 45 7F 77 DB 39 02 DE 87 53 D2 96 3B 89 F1 1A
Mix Column:	58 1B DB 1B 4D 4B E7 6B CA 5A CA B0 F1 AC A8 E5
Add Round Key:	AA 61 82 68 8F DD D2 32 5F E3 4A 46 03 EF D2 9A
Round 3	
Key:	3D 47 1E 6D 80 16 23 7A 47 FE 7E 88 7D 3E 44 3B
Subbyte:	AC EF 13 45 73 C1 B5 23 CF 11 D6 5A 7B DF B5 B8
Shift Rows:	AC EF 13 45 C1 B5 23 73 D6 5A CF 11 B8 7B DF B5
Mix Column:	75 20 53 BB EC 0B C0 25 09 63 CF D0 93 33 7C DC
Add Round Key:	48 67 4D D6 6C 1D E3 5F 4E 9D B1 58 EE 0D 38 E7
Round 4	
Key:	EF A8 B6 DB 44 52 71 0B A5 5B 25 AD 41 7F 3B 00
Subbyte:	52 85 E3 F6 50 A4 11 CF 2F 5E C8 6A 28 D7 07 94
Shift Rows:	52 85 E3 F6 A4 11 CF 50 C8 6A 2F 5E 94 28 D7 07
Mix Column:	0F 60 6F 5E D6 31 C0 B3 DA 38 10 13 A9 BF 6B 01
Add Round Key:	E0 C8 D9 85 92 63 B1 B8 7F 63 35 BE E8 C0 50 01
Round 5	
Key:	D4 7C CA 11 D1 83 F2 F9 C6 9D B8 15 F8 87 BC BC
Subbyte:	5B 54 D2 2F 49 53 16 AA 84 53 A5 42 F5 B9 1E 7C
Shift Rows:	5B 54 D2 2F 53 16 AA 49 A5 42 84 53 7C F5 B9 1E
Mix Column:	9A 25 67 C8 AD 4B B3 56 05 1A BB E2 E3 81 2A 57
Add Round Key:	4E 59 AD D9 7C C8 41 AF C3 87 03 F7 1B 06 96 EB

```
Round 6
Key:                6D 11 DB CA 88 0B F9 00 A3 3E 86 93 7A FD 41 FD
Subbyte:            2F CB 95 35 10 E8 83 79 2E 17 7B 68 AF 6F 90 E9
Shift Rows:         2F CB 95 35 E8 83 79 10 7B 68 2E 17 E9 AF 6F 90
Mix Column:         EF D4 FB DD 80 C1 7A BC 11 72 01 A0 2B E8 2D 63
Add Round Key:      82 C5 20 17 08 CA 83 BC B2 4C 87 33 51 15 6C 9E

Round 7
Key:                4E 5F 84 4E 54 5F A6 A6 F7 C9 4F DC 0E F3 B2 4F
Subbyte:            13 A6 B7 F0 30 74 EC 65 37 29 17 C3 D1 59 50 0B
Shift Rows:         13 A6 B7 F0 74 EC 65 30 17 C3 37 29 0B D1 59 50
Mix Column:         A6 6A B4 D2 C9 EA 7D BB 54 BF 57 62 40 67 22 B2
Add Round Key:      E8 35 30 9C 9D B5 DB 1D A3 76 18 BE 4E 94 90 FD

Round 8
Key:                EA B5 31 7F D2 8D 2B 8D 73 BA F5 29 21 D2 60 2F
Subbyte:            9B 96 04 DE 5E D5 B9 A4 0A 38 AD AE 2F 22 60 54
Shift Rows:         9B 96 04 DE D5 B9 A4 5E AD AE 0A 38 54 2F 22 60
Mix Column:         B0 66 D7 1D 92 39 6B 4A F3 19 D2 50 66 E8 E6 DF
Add Round Key:      5A D3 E6 62 40 B4 40 C7 80 A3 27 79 47 3A 86 F0

Round 9
Key:                AC 19 28 57 77 FA D1 5C 66 DC 29 00 F3 21 41 6E
Subbyte:            BE 66 8E AA 09 8D 09 C6 CD 0A CC B6 A0 80 44 8C
Shift Rows:         BE 66 8E AA 8D 09 C6 09 CC B6 CD 0A 8C A0 80 44
Mix Column:         AB C1 1B 1A 7C 15 D5 E2 3F E3 52 7B 9B 4E 99 6E
Add Round Key:      07 D8 33 4D 0B EF 04 BE 59 3F 7B 7B 68 6F D8 00

Round 10
Key:                D0 C9 E1 B6 14 EE 3F 63 F9 25 0C 0C A8 89 C8 A6
Subbyte:            C5 61 C3 E3 2B DF F2 AE CB 75 21 21 45 A8 61 63
Shift Rows:         C5 61 C3 E3 DF F2 AE 2B 21 21 CB 75 63 45 A8 61
Add Round Key:      15 A8 22 55 CB 1C 91 48 D8 04 C7 79 CB CC 60 C7

Cipher Text:        15 CB D8 CB A8 1C 04 CC 22 91 C7 60 55 48 79 C7
```

4.4 Example using random sequences of S-Boxes for both algorithms

Finally, by using a ranomzied sequence to select the S-Boxes to be used in each round, the Fortified AES provides another completely different cipher text from those previously seen in this paper.

Figure 16: Sequence and results using random sequence of S-Boxes for both algorithms

	Round										
Algorithm	I	1	2	3	4	5	6	7	8	9	F
Key Expansion	27	25	9	24	18	27	10	14	8	30	23
Rijndael	24	26	11	4	27	15	12	24	15	23	4

```
Plain Text:          32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34

Cipher Key:          2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

Initial Block:       32 88 31 E0 43 5A 31 37 F6 30 98 07 A8 8D A2 34

Initial Round
Key:                 2B 28 AB 09 7E AE F7 CF 15 D2 15 4F 16 A6 88 3C
Add round key:       19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08

Round 1
Key:                 C2 EA 41 48 E5 4B BC 73 40 92 87 C8 4C EA 62 5E
Subbyte:             AB 10 1D 5C A6 12 D6 6A 78 CF 5E 2F 86 30 43 B9
Shift Rows:          AB 10 1D 5C 12 D6 6A A6 5E 2F 78 CF B9 86 30 43
Mix Column:          5A 2E CC 03 D4 96 71 02 13 CF D7 7C C3 18 55 0B
Add Round Key:       98 C4 8D 4B 31 DD CD 71 53 5D 50 B4 8F F2 37 55

Round 2
Key:                 96 7C 3D 75 C5 8E 32 41 2D BF 38 F0 0B E1 83 DD
Subbyte:             08 AE BB AF ED D8 95 FF E6 CB FE 5D 87 D2 D3 B4
Shift Rows:          08 AE BB AF D8 95 FF ED FE 5D E6 CB B4 87 D2 D3
Mix Column:          5B 39 43 71 0E 8D BD FB F0 61 FE A1 3F 34 70 71
Add Round Key:       CD 45 7E 04 CB 03 8F BA DD DE C6 51 34 D5 F3 AC
```

Round 3

Key:	04 78 45 30 01 8F BD FC DC 63 5B AB 58 B9 3A E7
Subbyte:	FC 71 EF 7D 9D B4 BB E3 95 DA FD CC C6 F3 F9 D5
Shift Rows:	FC 71 EF 7D B4 BB E3 9D FD CC 95 DA D5 C6 F3 F9
Mix Column:	0C 08 9D 53 46 95 65 D0 CD 18 33 5F E7 45 A1 1F
Add Round Key:	08 70 D8 63 47 1A D8 2C 11 7B 68 F4 BF FC 9B F8

Round 4

Key:	7B 03 46 76 DE 51 EC 10 F4 97 CC 67 64 DD E7 00
Subbyte:	C7 59 DF BC 37 CF DF D0 D4 B1 32 D2 D9 56 80 0D
Shift Rows:	C7 59 DF BC CF DF D0 37 32 D2 D4 B1 0D D9 56 80
Mix Column:	E0 3F 4C F7 E5 48 55 66 7B 49 BA 69 49 B3 2E 42
Add Round Key:	9B 3C 0A 81 3B 19 B9 76 8F DE 76 0E 2D 6E C9 42

Round 5

Key:	AE AD EB 9D CC 9D 71 61 97 00 CC AB BC 61 86 86
Subbyte:	8D CC D4 58 5A CA F4 19 FF DC 19 1B D9 78 F5 8A
Shift Rows:	8D CC D4 58 CA F4 19 5A 19 1B FF DC 8A D9 78 F5
Mix Column:	D7 46 83 77 3F 57 18 FA 6C E2 3C A5 50 09 ED 03
Add Round Key:	79 EB 68 EA F3 CA 69 9B FB E2 F0 0E EC 68 6B 85

Round 6

Key:	A1 0C E7 7A 34 A9 D8 B9 DF DF 13 B8 99 F8 7E F8
Subbyte:	09 58 21 5B 3C 8F B1 E2 E4 4F 20 85 98 21 E8 ED
Shift Rows:	09 58 21 5B 8F B1 E2 3C 20 85 E4 4F ED 98 21 E8
Mix Column:	3F 0F D0 55 EB 2D E8 1A 80 4B 19 B0 1F 9D 27 3F
Add Round Key:	9E 03 37 2F DF 84 30 A3 5F 94 0A 08 86 65 59 C7

Round 7

Key:	C8 C4 23 59 E7 4E 96 2F 2D F2 E1 59 73 8B F5 0D
Subbyte:	A6 38 87 C9 AF B4 4D 29 31 1F 56 62 3A 1B 3F 88
Shift Rows:	A6 38 87 C9 B4 4D 29 AF 56 62 31 1F 88 3A 1B 3F
Mix Column:	4E FF 90 43 73 3E 9D 46 E9 FF E1 19 18 13 68 5A
Add Round Key:	86 3B B3 1A 94 70 0B 69 C4 0D 00 40 6B 98 9D 57

Round 8

Key:	65 A1 82 DB 2D 63 F5 DA 1E EC 0D 54 B9 32 C7 CA
Subbyte:	06 5A D5 0D E5 40 8B F6 29 BF 63 8C FE 61 CE 73
Shift Rows:	06 5A D5 0D 40 8B F6 E5 63 8C 29 BF 73 FE 61 CE
Mix Column:	DC DC F8 C3 50 26 A4 C8 15 CB D2 C4 CF 92 E5 56
Add Round Key:	B9 7D 7A 18 7D 45 51 12 0B 27 DF 90 76 A0 22 9C

Round 9

```
Key:                CC 6D EF 34 0C 6F 9A 40 98 74 79 2D E9 DB 1C D6
Subbyte:            89 72 9A 1A 72 7B D6 60 24 0F 37 9E F1 DD FE 78
Shift Rows:         89 72 9A 1A 7B D6 60 72 37 9E 24 0F 78 F1 DD FE
Mix Column:         13 32 AE 53 5E 8D EB 11 14 8B 16 B7 E4 FF 50 6C
Add Round Key:      DF 5F 41 67 52 E2 71 51 8C FF 6F 9A 0D 24 4C BA

Round 10
Key:                80 ED 02 36 BC D3 49 09 C4 B0 C9 E4 D2 09 15 C3
Subbyte:            CD 74 49 34 13 46 29 CC 76 99 77 20 F5 B0 DB E3
Shift Rows:         CD 74 49 34 46 29 CC 13 77 20 76 99 E3 F5 B0 DB
Add Round Key:      4D 99 4B 02 FA FA 85 1A B3 90 BF 7D 31 FC A5 18

Cipher Text:        4D FA B3 31 99 FA 90 FC 4B 85 BF A5 02 1A 7D 18
```

The 128 bit AES algorithm has a keyspace 2^{128} (3.4^{38}) potential keys and the same number of potential cipher texts for a given input, while using the same number of rounds for transformations the Fortified AES expands the keyspace by 30^{20} based on the number of permutations afforded each sequence. This results in an overall potential ciphertexts of $2^{128} * 30^{20}$ (1.19^{68}) representing an increased keyspace.

Chapter 5: Conclusion

AES, as the industry standard for symmetric encryption, has been called into question due to quantum computing. To that end, research within the field seeks to bolster and fortify the process, in part due to external algorithms reducing the time needed to search keyspace (e.g., Grover). This research answers the question 'Can user-selected S-Boxes fortify AES encryption by increasing keyspace?' by examining AES and how a fortified extension can support increased security by expanding keyspace, concluding yes, given the proposed code demonstration. The increase in security is achieved by implementing a feature, which allows users to fortify the AES process by applying a unique S-Box sequence across rounds. This is distinctly different from traditional methodology, which requires a user to use a single S-Box throughout the process. The fortified AES is particularly advantageous as the fortified method requires no additional rounds and still adds to the keyspace. While traditional methods require additional rounds of transformation, this research provides an example of how the resources can be spent differently by allowing users to select a sequence of S-Box. By allowing this choice, the algorithm further confuses the relationship between the message and the ciphertext, providing additional security.

This paper provides empirical evidence for a method of AES fortification, however, some limitations do exist. One reason that S-Boxes are currently a mainstay of symmetric encryption is the high number of permutations (n=256!) available. However, within that number, not all S-Boxes have an associated inverse and not all S-Boxes are the same strength. First, an invertible S-Box is required to reverse a substitution operation. Without being able to reverse the substitution a message can not be recovered from the ciphertext. Second, S-Boxes vary in strength, in fact, the use of some S-Boxes might introduce weaknesses into the ciphertext message. A weak S-Box does not sufficiently confuse the relationship between the ciphertext and

plain text. This can allow cryptanalysis techniques to deduce the relationship within the encryption. This research accounts for both of these issues by first introducing a verification algorithm (p.17) to ensure an S-Box with an inverse is chosen. Second, in order to ensure only strong S-boxes were chosen this thesis uses only S-Boxes generated using the affine transformation and irreversible polynomials of degree 8 (p.15).

Appendix A: Ciphertext to Plaintext Examples

A.1 Standard AES

```
Encrypted Text:       39 02 DC 19 25 DC 11 6A 84 09 85 0B 1D FB 97 32

Cipher Key:           2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

Initial Block:        39 02 DC 19 25 DC 11 6A 84 09 85 0B 1D FB 97 32

Round 1
Key:                  D0 C9 E1 B6 14 EE 3F 63 F9 25 0C 0C A8 89 C8 A6
Add Round Key:        E9 CB 3D AF 31 32 2E 09 7D 2C 89 07 B5 72 5F 94
Inverse Shift Rows:   E9 CB 3D AF 09 31 32 2E 89 07 7D 2C 72 5F 94 B5
Inverse Subbyte:      EB 59 8B 1B 40 2E A1 C3 F2 38 13 42 1E 84 E7 D2

Round 2
Key:                  AC 19 28 57 77 FA D1 5C 66 DC 29 00 F3 21 41 6E
Add Round Key:        47 40 A3 4C 37 D4 70 9F 94 E4 3A 42 ED A5 A6 BC
Inverse Mix Column:   87 F2 4D 97 6E 4C 90 EC 46 E7 4A C3 A6 8C D8 95
Inverse Shift Rows:   87 F2 4D 97 EC 6E 4C 90 4A C3 46 E7 8C D8 95 A6
Inverse Subbyte:      EA 04 65 85 83 45 5D 96 5C 33 98 B0 F0 2D AD C5

Round 3
Key:                  EA B5 31 7F D2 8D 2B 8D 73 BA F5 29 21 D2 60 2F
Add Round Key:        00 B1 54 FA 51 C8 76 1B 2F 89 6D 99 D1 FF CD EA
Inverse Mix Column:   BE D4 0A DA 3B E1 64 83 D4 F2 2C 86 FE C8 C0 4D
Inverse Shift Rows:   BE D4 0A DA 83 3B E1 64 2C 86 D4 F2 C8 C0 4D FE
Inverse Subbyte:      5A 19 A3 7A 41 49 E0 8C 42 DC 19 04 B1 1F 65 0C

Round 4
Key:                  4E 5F 84 4E 54 5F A6 A6 F7 C9 4F DC 0E F3 B2 4F
Add Round Key:        14 46 27 34 15 16 46 2A B5 15 56 D8 BF EC D7 43
Inverse Mix Column:   F7 27 9B 54 83 43 B5 AB 40 3D 31 A9 3F F0 FF D3
Inverse Shift Rows:   F7 27 9B 54 AB 83 43 B5 31 A9 40 3D F0 FF D3 3F
Inverse Subbyte:      26 3D E8 FD 0E 41 64 D2 2E B7 72 8B 17 7D A9 25

Round 5
Key:                  6D 11 DB CA 88 0B F9 00 A3 3E 86 93 7A FD 41 FD
Add Round Key:        4B 2C 33 37 86 4A 9D D2 8D 89 F4 18 6D 80 E8 D8
Inverse Mix Column:   A1 78 10 4C 4F E8 D5 63 3D 03 A8 29 FE FC DF 23
Inverse Shift Rows:   A1 78 10 4C 63 4F E8 D5 A8 29 3D 03 FC DF 23 FE
Inverse Subbyte:      F1 C1 7C 5D 00 92 C8 B5 6F 4C 8B D5 55 EF 32 0C
```

```
Round 6
Key:                  D4 7C CA 11 D1 83 F2 F9 C6 9D B8 15 F8 87 BC BC
Add Round Key:        25 BD B6 4C D1 11 3A 4C A9 D1 33 C0 AD 68 8E B0
Inverse Mix Column:   E1 E8 35 97 FB C8 6C 4F 96 AE D2 FB 7C 9B BA 53
Inverse Shift Rows:   E1 E8 35 97 4F FB C8 6C D2 FB 96 AE 9B BA 53 7C
Inverse Subbyte:      E0 C8 D9 85 92 63 B1 B8 7F 63 35 BE E8 C0 50 01

Round 7
Key:                  EF A8 B6 DB 44 52 71 0B A5 5B 25 AD 41 7F 3B 00
Add Round Key:        0F 60 6F 5E D6 31 C0 B3 DA 38 10 13 A9 BF 6B 01
Inverse Mix Column:   52 85 E3 F6 A4 11 CF 50 C8 6A 2F 5E 94 28 D7 07
Inverse Shift Rows:   52 85 E3 F6 50 A4 11 CF 2F 5E C8 6A 28 D7 07 94
Inverse Subbyte:      48 67 4D D6 6C 1D E3 5F 4E 9D B1 58 EE 0D 38 E7

Round 8
Key:                  3D 47 1E 6D 80 16 23 7A 47 FE 7E 88 7D 3E 44 3B
Add Round Key:        75 20 53 BB EC 0B C0 25 09 63 CF D0 93 33 7C DC
Inverse Mix Column:   AC EF 13 45 C1 B5 23 73 D6 5A CF 11 B8 7B DF B5
Inverse Shift Rows:   AC EF 13 45 73 C1 B5 23 CF 11 D6 5A 7B DF B5 B8
Inverse Subbyte:      AA 61 82 68 8F DD D2 32 5F E3 4A 46 03 EF D2 9A

Round 9
Key:                  F2 7A 59 73 C2 96 35 59 95 B9 80 F6 F2 43 7A 7F
Add Round Key:        58 1B DB 1B 4D 4B E7 6B CA 5A CA B0 F1 AC A8 E5
Inverse Mix Column:   49 45 7F 77 DB 39 02 DE 87 53 D2 96 3B 89 F1 1A
Inverse Shift Rows:   49 45 7F 77 DE DB 39 02 D2 96 87 53 89 F1 1A 3B
Inverse Subbyte:      A4 68 6B 02 9C 9F 5B 6A 7F 35 EA 50 F2 2B 43 49

Round 10
Key:                  A0 88 23 2A FA 54 A3 6C FE 2C 39 76 17 B1 39 05
Add Round Key:        04 E0 48 28 66 CB F8 06 81 19 D3 26 E5 9A 7A 4C
Inverse Mix Column:   D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5
Inverse Shift Rows:   D4 E0 B8 1E 27 BF B4 41 11 98 5D 52 AE F1 E5 30
Inverse Subbyte:      19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08

Last Round
Key:                  2B 28 AB 09 7E AE F7 CF 15 D2 15 4F 16 A6 88 3C
Add Round Key:        32 88 31 E0 43 5A 31 37 F6 30 98 07 A8 8D A2 34
Decrypted Text:       32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
```

A.2 Example 2 S-Box substitution in Key Expansion

```
Encrypted Text:       A2 BC C4 6C 10 62 EE F3 31 06 4F F2 9F 31 47 AC

Cipher Key:           2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

Initial Block:        A2 BC C4 6C 10 62 EE F3 31 06 4F F2 9F 31 47 AC

Round 1
Key:                  39 B6 AD 6C 19 29 E6 70 70 7C AA 7A 26 82 6E 85
Add Round Key:        9B 0A 69 00 09 4B 08 83 41 7A E5 88 B9 B3 29 29
Inverse Shift Rows:   9B 0A 69 00 83 09 4B 08 E5 88 41 7A B3 29 29 B9
Inverse Subbyte:      E8 A3 E4 52 41 40 CC BF 2A 97 F8 BD 4B 4C 4C DB

Round 2
Key:                  9F 8F 1B C1 69 30 CF 96 99 0C D6 D0 5E A4 EC EB
Add Round Key:        77 2C FF 93 28 70 03 29 B3 9B 2E 6D 15 E8 A0 30
Inverse Mix Column:   84 32 BA B7 81 A7 2C 5B A9 DB 8F 9C 55 61 6B 97
Inverse Shift Rows:   84 32 BA B7 5B 81 A7 2C 8F 9C A9 DB 61 6B 97 55
Inverse Subbyte:      4F A1 C0 20 57 91 89 42 73 1C B7 9F D8 05 85 ED

Round 3
Key:                  4F 10 94 DA 06 59 FF 59 5C 95 DA 06 09 FA 48 07
Add Round Key:        00 B1 54 FA 51 C8 76 1B 2F 89 6D 99 D1 FF CD EA
Inverse Mix Column:   BE D4 0A DA 3B E1 64 83 D4 F2 2C 86 FE C8 C0 4D
Inverse Shift Rows:   BE D4 0A DA 83 3B E1 64 2C 86 D4 F2 C8 C0 4D FE
Inverse Subbyte:      5A 19 A3 7A 41 49 E0 8C 42 DC 19 04 B1 1F 65 0C

Round 4
Key:                  4E 5F 84 4E 54 5F A6 A6 F7 C9 4F DC 0E F3 B2 4F
Add Round Key:        14 46 27 34 15 16 46 2A B5 15 56 D8 BF EC D7 43
Inverse Mix Column:   F7 27 9B 54 83 43 B5 AB 40 3D 31 A9 3F F0 FF D3
Inverse Shift Rows:   F7 27 9B 54 AB 83 43 B5 31 A9 40 3D F0 FF D3 3F
Inverse Subbyte:      26 3D E8 FD 0E 41 64 D2 2E B7 72 8B 17 7D A9 25

Round 5
Key:                  6D 11 DB CA 88 0B F9 00 A3 3E 86 93 7A FD 41 FD
Add Round Key:        4B 2C 33 37 86 4A 9D D2 8D 89 F4 18 6D 80 E8 D8
Inverse Mix Column:   A1 78 10 4C 4F E8 D5 63 3D 03 A8 29 FE FC DF 23
Inverse Shift Rows:   A1 78 10 4C 63 4F E8 D5 A8 29 3D 03 FC DF 23 FE
Inverse Subbyte:      F1 C1 7C 5D 00 92 C8 B5 6F 4C 8B D5 55 EF 32 0C

Round 6
Key:                  D4 7C CA 11 D1 83 F2 F9 C6 9D B8 15 F8 87 BC BC
```

```
Add Round Key:         25 BD B6 4C D1 11 3A 4C A9 D1 33 C0 AD 68 8E B0
Inverse Mix Column:    E1 E8 35 97 FB C8 6C 4F 96 AE D2 FB 7C 9B BA 53
Inverse Shift Rows:    E1 E8 35 97 4F FB C8 6C D2 FB 96 AE 9B BA 53 7C
Inverse Subbyte:       E0 C8 D9 85 92 63 B1 B8 7F 63 35 BE E8 C0 50 01

Round 7
Key:                   EF A8 B6 DB 44 52 71 0B A5 5B 25 AD 41 7F 3B 00
Add Round Key:         0F 60 6F 5E D6 31 C0 B3 DA 38 10 13 A9 BF 6B 01
Inverse Mix Column:    52 85 E3 F6 A4 11 CF 50 C8 6A 2F 5E 94 28 D7 07
Inverse Shift Rows:    52 85 E3 F6 50 A4 11 CF 2F 5E C8 6A 28 D7 07 94
Inverse Subbyte:       48 67 4D D6 6C 1D E3 5F 4E 9D B1 58 EE 0D 38 E7

Round 8
Key:                   3D 47 1E 6D 80 16 23 7A 47 FE 7E 88 7D 3E 44 3B
Add Round Key:         75 20 53 BB EC 0B C0 25 09 63 CF D0 93 33 7C DC
Inverse Mix Column:    AC EF 13 45 C1 B5 23 73 D6 5A CF 11 B8 7B DF B5
Inverse Shift Rows:    AC EF 13 45 73 C1 B5 23 CF 11 D6 5A 7B DF B5 B8
Inverse Subbyte:       AA 61 82 68 8F DD D2 32 5F E3 4A 46 03 EF D2 9A

Round 9
Key:                   F2 7A 59 73 C2 96 35 59 95 B9 80 F6 F2 43 7A 7F
Add Round Key:         58 1B DB 1B 4D 4B E7 6B CA 5A CA B0 F1 AC A8 E5
Inverse Mix Column:    49 45 7F 77 DB 39 02 DE 87 53 D2 96 3B 89 F1 1A
Inverse Shift Rows:    49 45 7F 77 DE DB 39 02 D2 96 87 53 89 F1 1A 3B
Inverse Subbyte:       A4 68 6B 02 9C 9F 5B 6A 7F 35 EA 50 F2 2B 43 49

Round 10
Key:                   A0 88 23 2A FA 54 A3 6C FE 2C 39 76 17 B1 39 05
Add Round Key:         04 E0 48 28 66 CB F8 06 81 19 D3 26 E5 9A 7A 4C
Inverse Mix Column:    D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5
Inverse Shift Rows:    D4 E0 B8 1E 27 BF B4 41 11 98 5D 52 AE F1 E5 30
Inverse Subbyte:       19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08

Last Round
Key:                   2B 28 AB 09 7E AE F7 CF 15 D2 15 4F 16 A6 88 3C
Add Round Key:         32 88 31 E0 43 5A 31 37 F6 30 98 07 A8 8D A2 34
Decrypted Text:        32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
```

A.3 S-Box substitution in Rijndael

```
Encrypted Text:       15 A8 22 55 CB 1C 91 48 D8 04 C7 79 CB CC 60 C7

Cipher Key:           2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

Initial Block:        15 A8 22 55 CB 1C 91 48 D8 04 C7 79 CB CC 60 C7

Round 1
Key:                  D0 C9 E1 B6 14 EE 3F 63 F9 25 0C 0C A8 89 C8 A6
Add Round Key:        C5 61 C3 E3 DF F2 AE 2B 21 21 CB 75 63 45 A8 61
Inverse Shift Rows:   C5 61 C3 E3 2B DF F2 AE CB 75 21 21 45 A8 61 63
Inverse Subbyte:      07 D8 33 4D 0B EF 04 BE 59 3F 7B 7B 68 6F D8 00

Round 2
Key:                  AC 19 28 57 77 FA D1 5C 66 DC 29 00 F3 21 41 6E
Add Round Key:        AB C1 1B 1A 7C 15 D5 E2 3F E3 52 7B 9B 4E 99 6E
Inverse Mix Column:   BE 66 8E AA 8D 09 C6 09 CC B6 CD 0A 8C A0 80 44
Inverse Shift Rows:   BE 66 8E AA 09 8D 09 C6 CD 0A CC B6 A0 80 44 8C
Inverse Subbyte:      5A D3 E6 62 40 B4 40 C7 80 A3 27 79 47 3A 86 F0

Round 3
Key:                  EA B5 31 7F D2 8D 2B 8D 73 BA F5 29 21 D2 60 2F
Add Round Key:        B0 66 D7 1D 92 39 6B 4A F3 19 D2 50 66 E8 E6 DF
Inverse Mix Column:   9B 96 04 DE D5 B9 A4 5E AD AE 0A 38 54 2F 22 60
Inverse Shift Rows:   9B 96 04 DE 5E D5 B9 A4 0A 38 AD AE 2F 22 60 54
Inverse Subbyte:      E8 35 30 9C 9D B5 DB 1D A3 76 18 BE 4E 94 90 FD

Round 4
Key:                  4E 5F 84 4E 54 5F A6 A6 F7 C9 4F DC 0E F3 B2 4F
Add Round Key:        A6 6A B4 D2 C9 EA 7D BB 54 BF 57 62 40 67 22 B2
Inverse Mix Column:   13 A6 B7 F0 74 EC 65 30 17 C3 37 29 0B D1 59 50
Inverse Shift Rows:   13 A6 B7 F0 30 74 EC 65 37 29 17 C3 D1 59 50 0B
Inverse Subbyte:      82 C5 20 17 08 CA 83 BC B2 4C 87 33 51 15 6C 9E

Round 5
Key:                  6D 11 DB CA 88 0B F9 00 A3 3E 86 93 7A FD 41 FD
Add Round Key:        EF D4 FB DD 80 C1 7A BC 11 72 01 A0 2B E8 2D 63
Inverse Mix Column:   2F CB 95 35 E8 83 79 10 7B 68 2E 17 E9 AF 6F 90
Inverse Shift Rows:   2F CB 95 35 10 E8 83 79 2E 17 7B 68 AF 6F 90 E9
Inverse Subbyte:      4E 59 AD D9 7C C8 41 AF C3 87 03 F7 1B 06 96 EB
```

```
Round 6
Key:                  D4 7C CA 11 D1 83 F2 F9 C6 9D B8 15 F8 87 BC BC
Add Round Key:        9A 25 67 C8 AD 4B B3 56 05 1A BB E2 E3 81 2A 57
Inverse Mix Column:   5B 54 D2 2F 53 16 AA 49 A5 42 84 53 7C F5 B9 1E
Inverse Shift Rows:   5B 54 D2 2F 49 53 16 AA 84 53 A5 42 F5 B9 1E 7C
Inverse Subbyte:      E0 C8 D9 85 92 63 B1 B8 7F 63 35 BE E8 C0 50 01

Round 7
Key:                  EF A8 B6 DB 44 52 71 0B A5 5B 25 AD 41 7F 3B 00
Add Round Key:        0F 60 6F 5E D6 31 C0 B3 DA 38 10 13 A9 BF 6B 01
Inverse Mix Column:   52 85 E3 F6 A4 11 CF 50 C8 6A 2F 5E 94 28 D7 07
Inverse Shift Rows:   52 85 E3 F6 50 A4 11 CF 2F 5E C8 6A 28 D7 07 94
Inverse Subbyte:      48 67 4D D6 6C 1D E3 5F 4E 9D B1 58 EE 0D 38 E7

Round 8
Key:                  3D 47 1E 6D 80 16 23 7A 47 FE 7E 88 7D 3E 44 3B
Add Round Key:        75 20 53 BB EC 0B C0 25 09 63 CF D0 93 33 7C DC
Inverse Mix Column:   AC EF 13 45 C1 B5 23 73 D6 5A CF 11 B8 7B DF B5
Inverse Shift Rows:   AC EF 13 45 73 C1 B5 23 CF 11 D6 5A 7B DF B5 B8
Inverse Subbyte:      AA 61 82 68 8F DD D2 32 5F E3 4A 46 03 EF D2 9A

Round 9
Key:                  F2 7A 59 73 C2 96 35 59 95 B9 80 F6 F2 43 7A 7F
Add Round Key:        58 1B DB 1B 4D 4B E7 6B CA 5A CA B0 F1 AC A8 E5
Inverse Mix Column:   49 45 7F 77 DB 39 02 DE 87 53 D2 96 3B 89 F1 1A
Inverse Shift Rows:   49 45 7F 77 DE DB 39 02 D2 96 87 53 89 F1 1A 3B
Inverse Subbyte:      A4 68 6B 02 9C 9F 5B 6A 7F 35 EA 50 F2 2B 43 49

Round 10
Key:                  A0 88 23 2A FA 54 A3 6C FE 2C 39 76 17 B1 39 05
Add Round Key:        04 E0 48 28 66 CB F8 06 81 19 D3 26 E5 9A 7A 4C
Inverse Mix Column:   D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5
Inverse Shift Rows:   D4 E0 B8 1E 27 BF B4 41 11 98 5D 52 AE F1 E5 30
Inverse Subbyte:      19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08

Last Round
Key:                  2B 28 AB 09 7E AE F7 CF 15 D2 15 4F 16 A6 88 3C
Add Round Key:        32 88 31 E0 43 5A 31 37 F6 30 98 07 A8 8D A2 34
Decrypted Text:       32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
```

A.4 Example using random sequences of S-Boxes for both algorithms

```
Encrypted Text:       4D 99 4B 02 FA FA 85 1A B3 90 BF 7D 31 FC A5 18

Cipher Key:           2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

Initial Block:        4D 99 4B 02 FA FA 85 1A B3 90 BF 7D 31 FC A5 18

Round 1
Key:                  80 ED 02 36 BC D3 49 09 C4 B0 C9 E4 D2 09 15 C3
Add Round Key:        CD 74 49 34 46 29 CC 13 77 20 76 99 E3 F5 B0 DB
Inverse Shift Rows:   CD 74 49 34 13 46 29 CC 76 99 77 20 F5 B0 DB E3
Inverse Subbyte:      DF 5F 41 67 52 E2 71 51 8C FF 6F 9A 0D 24 4C BA

Round 2
Key:                  CC 6D EF 34 0C 6F 9A 40 98 74 79 2D E9 DB 1C D6
Add Round Key:        13 32 AE 53 5E 8D EB 11 14 8B 16 B7 E4 FF 50 6C
Inverse Mix Column:   89 72 9A 1A 7B D6 60 72 37 9E 24 0F 78 F1 DD FE
Inverse Shift Rows:   89 72 9A 1A 72 7B D6 60 24 0F 37 9E F1 DD FE 78
Inverse Subbyte:      B9 7D 7A 18 7D 45 51 12 0B 27 DF 90 76 A0 22 9C

Round 3
Key:                  65 A1 82 DB 2D 63 F5 DA 1E EC 0D 54 B9 32 C7 CA
Add Round Key:        DC DC F8 C3 50 26 A4 C8 15 CB D2 C4 CF 92 E5 56
Inverse Mix Column:   06 5A D5 0D 40 8B F6 E5 63 8C 29 BF 73 FE 61 CE
Inverse Shift Rows:   06 5A D5 0D E5 40 8B F6 29 BF 63 8C FE 61 CE 73
Inverse Subbyte:      86 3B B3 1A 94 70 0B 69 C4 0D 00 40 6B 98 9D 57

Round 4
Key:                  C8 C4 23 59 E7 4E 96 2F 2D F2 E1 59 73 8B F5 0D
Add Round Key:        4E FF 90 43 73 3E 9D 46 E9 FF E1 19 18 13 68 5A
Inverse Mix Column:   A6 38 87 C9 B4 4D 29 AF 56 62 31 1F 88 3A 1B 3F
Inverse Shift Rows:   A6 38 87 C9 AF B4 4D 29 31 1F 56 62 3A 1B 3F 88
Inverse Subbyte:      9E 03 37 2F DF 84 30 A3 5F 94 0A 08 86 65 59 C7

Round 5
Key:                  A1 0C E7 7A 34 A9 D8 B9 DF DF 13 B8 99 F8 7E F8
Add Round Key:        3F 0F D0 55 EB 2D E8 1A 80 4B 19 B0 1F 9D 27 3F
Inverse Mix Column:   09 58 21 5B 8F B1 E2 3C 20 85 E4 4F ED 98 21 E8
Inverse Shift Rows:   09 58 21 5B 3C 8F B1 E2 E4 4F 20 85 98 21 E8 ED
Inverse Subbyte:      79 EB 68 EA F3 CA 69 9B FB E2 F0 0E EC 68 6B 85

Round 6
Key:                  AE AD EB 9D CC 9D 71 61 97 00 CC AB BC 61 86 86
Add Round Key:        D7 46 83 77 3F 57 18 FA 6C E2 3C A5 50 09 ED 03
```

```
Inverse Mix Column:   8D CC D4 58 CA F4 19 5A 19 1B FF DC 8A D9 78 F5
Inverse Shift Rows:   8D CC D4 58 5A CA F4 19 FF DC 19 1B D9 78 F5 8A
Inverse Subbyte:      9B 3C 0A 81 3B 19 B9 76 8F DE 76 0E 2D 6E C9 42

Round 7
Key:                  7B 03 46 76 DE 51 EC 10 F4 97 CC 67 64 DD E7 00
Add Round Key:        E0 3F 4C F7 E5 48 55 66 7B 49 BA 69 49 B3 2E 42
Inverse Mix Column:   C7 59 DF BC CF DF D0 37 32 D2 D4 B1 0D D9 56 80
Inverse Shift Rows:   C7 59 DF BC 37 CF DF D0 D4 B1 32 D2 D9 56 80 0D
Inverse Subbyte:      08 70 D8 63 47 1A D8 2C 11 7B 68 F4 BF FC 9B F8

Round 8
Key:                  04 78 45 30 01 8F BD FC DC 63 5B AB 58 B9 3A E7
Add Round Key:        0C 08 9D 53 46 95 65 D0 CD 18 33 5F E7 45 A1 1F
Inverse Mix Column:   FC 71 EF 7D B4 BB E3 9D FD CC 95 DA D5 C6 F3 F9
Inverse Shift Rows:   FC 71 EF 7D 9D B4 BB E3 95 DA FD CC C6 F3 F9 D5
Inverse Subbyte:      CD 45 7E 04 CB 03 8F BA DD DE C6 51 34 D5 F3 AC

Round 9
Key:                  96 7C 3D 75 C5 8E 32 41 2D BF 38 F0 0B E1 83 DD
Add Round Key:        5B 39 43 71 0E 8D BD FB F0 61 FE A1 3F 34 70 71
Inverse Mix Column:   08 AE BB AF D8 95 FF ED FE 5D E6 CB B4 87 D2 D3
Inverse Shift Rows:   08 AE BB AF ED D8 95 FF E6 CB FE 5D 87 D2 D3 B4
Inverse Subbyte:      98 C4 8D 4B 31 DD CD 71 53 5D 50 B4 8F F2 37 55

Round 10
Key:                  C2 EA 41 48 E5 4B BC 73 40 92 87 C8 4C EA 62 5E
Add Round Key:        5A 2E CC 03 D4 96 71 02 13 CF D7 7C C3 18 55 0B
Inverse Mix Column:   AB 10 1D 5C 12 D6 6A A6 5E 2F 78 CF B9 86 30 43
Inverse Shift Rows:   AB 10 1D 5C A6 12 D6 6A 78 CF 5E 2F 86 30 43 B9
Inverse Subbyte:      19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08

Last Round
Key:                  2B 28 AB 09 7E AE F7 CF 15 D2 15 4F 16 A6 88 3C
Add Round Key:        32 88 31 E0 43 5A 31 37 F6 30 98 07 A8 8D A2 34
Decrypted Text:       32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
```

Appendix B: Fortified AES Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace AES_Fortification
{
        public partial class Form1 : Form
        {

        // Global Variables

        Random rand = new Random();

        int[][] baseMatrix;
        int[][] invMatrix;

        int[] baseV = { 1, 0, 0, 0, 1, 1, 1, 1 };
        int[] invV = { 0, 0, 1, 0, 0, 1, 0, 1 };

        byte[] baseOffV = { 1, 1, 0, 0, 0, 1, 1, 0 };
        byte[] invOffV = { 1, 0, 1, 0, 0, 0, 0, 0 };

        int[][] mcMatrix;
        int[][] imcMatrix;

        int[] mcVector = { 2, 3, 1, 1 };
        int[] imcVector = { 14, 11, 13, 9 };

        int[] rcon;
        int[][] rndKeys;

        int[] polyList = { 0, 283, 285, 299, 301, 313, 319, 333, 351, 355, 357, 361, 369, 375,
379, 391, 395, 397, 415, 419, 425, 433, 445, 451, 463, 471, 477, 487, 499, 501, 505 };
        int[] rkSeq = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
        int[] sbSeq = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

        int[] cipKey = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF, 0xFE, 0xDC, 0xBA, 0x98,
0x76, 0x54, 0x32, 0x10 };

        int[] plnTxt = new int[16];

        int[][] sBoxes = new int[30][];
        int[][] isBoxes = new int[30][];


        // Initialization
        public Form1()
        {
                InitializeComponent();


                // generation of global variables
                baseMatrix = genMat(baseV);
```

```
        invMatrix = genMat(invV);
        mcMatrix = genMat(mcVector);
        imcMatrix = genMat(imcVector);
        rcon = genRcon();

        // generate static substituion boxes and inverse substitution boxes
        for (int i = 0; i < 30; i++)
        {
        sBoxes[i] = new int[256];
        isBoxes[i] = new int[256];

        for (int j = 0; j < 256; j++)
        {
                sBoxes[i][j] = genSubByte(j, polyList[i + 1]);
                isBoxes[i][j] = genInvSubByte(j, polyList[i + 1]);

        }
        }

        // set round key combo boxes to first item
        rkR0.SelectedIndex = 0;
        rkR1.SelectedIndex = 0;
        rkR2.SelectedIndex = 0;
        rkR3.SelectedIndex = 0;
        rkR4.SelectedIndex = 0;
        rkR5.SelectedIndex = 0;
        rkR6.SelectedIndex = 0;
        rkR7.SelectedIndex = 0;
        rkR8.SelectedIndex = 0;
        rkR9.SelectedIndex = 0;
        rkR10.SelectedIndex = 0;

        // set substitution combo boxes to first item
        sbR0.SelectedIndex = 0;
        sbR1.SelectedIndex = 0;
        sbR2.SelectedIndex = 0;
        sbR3.SelectedIndex = 0;
        sbR4.SelectedIndex = 0;
        sbR5.SelectedIndex = 0;
        sbR6.SelectedIndex = 0;
        sbR7.SelectedIndex = 0;
        sbR8.SelectedIndex = 0;
        sbR9.SelectedIndex = 0;
        sbR10.SelectedIndex = 0;

}

//#############################################################################################
private void runAlgBtn_Click(object sender, EventArgs e)
{
        // get plain text and cipher key
        for (int i = 0; i < 16; i++)
        {
        plnTxt[i] = Convert.ToByte(msgBox.Text.Substring(i * 2, 2), 16);
        cipKey[i] = Convert.ToByte(keyBox.Text.Substring(i * 2, 2), 16);
        }

        // get round key sequence
        rkSeq[0] = int.Parse(rkR0.SelectedItem.ToString());   // initial
        rkSeq[1] = int.Parse(rkR1.SelectedItem.ToString());   // round 1
        rkSeq[2] = int.Parse(rkR2.SelectedItem.ToString());   // round 2
        rkSeq[3] = int.Parse(rkR3.SelectedItem.ToString());   // round 3
```

```
        rkSeq[4] = int.Parse(rkR4.SelectedItem.ToString());   // round 4
        rkSeq[5] = int.Parse(rkR5.SelectedItem.ToString());   // round 5
        rkSeq[6] = int.Parse(rkR6.SelectedItem.ToString());   // round 6
        rkSeq[7] = int.Parse(rkR7.SelectedItem.ToString());   // round 7
        rkSeq[8] = int.Parse(rkR8.SelectedItem.ToString());   // round 8
        rkSeq[9] = int.Parse(rkR9.SelectedItem.ToString());   // round 9
        rkSeq[10] = int.Parse(rkR10.SelectedItem.ToString()); // round 10

        // get substitution sequence
        sbSeq[0] = int.Parse(sbR0.SelectedItem.ToString());   // initial
        sbSeq[1] = int.Parse(sbR1.SelectedItem.ToString());   // round 1
        sbSeq[2] = int.Parse(sbR2.SelectedItem.ToString());   // round 2
        sbSeq[3] = int.Parse(sbR3.SelectedItem.ToString());   // round 3
        sbSeq[4] = int.Parse(sbR4.SelectedItem.ToString());   // round 4
        sbSeq[5] = int.Parse(sbR5.SelectedItem.ToString());   // round 5
        sbSeq[6] = int.Parse(sbR6.SelectedItem.ToString());   // round 6
        sbSeq[7] = int.Parse(sbR7.SelectedItem.ToString());   // round 7
        sbSeq[8] = int.Parse(sbR8.SelectedItem.ToString());   // round 8
        sbSeq[9] = int.Parse(sbR9.SelectedItem.ToString());   // round 9
        sbSeq[10] = int.Parse(sbR10.SelectedItem.ToString()); // round 10

        // generate round keys
        rndKeys = roundKeys();

        // run encode
        int[][] encoded = encode(plnTxt, cipKey);

        // run decode
        int[][] decoded = decode(encoded[54], cipKey);

        writeResults(encoded, true);

        writeResults(decoded, false);

}

// Helper Functions
//##############################################################################################

private int[][] encode(int[] plnTxt, int[] cipKey)
{
        //
        int idx = 0;

        // initalize return array
        int[][] retEnc = new int[55][];

        for(int i = 0; i<55; i++)
        {
        retEnc[i] = new int[16];
        }



        // add plain text and cipher key to array
        retEnc[idx++] = plnTxt;
        retEnc[idx++] = cipKey;

        // initial block
        retEnc[idx++] = parBlk(plnTxt);

        // initial round
```

```
        // set round key
        retEnc[idx++] = parBlk(rndKeys[0]);

        // add round key
        retEnc[idx++] = addRound(parBlk(cipKey), parBlk(plnTxt));

        // 10 repeating rounds
        for(int i = 1; i < 10; i++)
        {
        // set round key
        retEnc[idx++] = parBlk(rndKeys[i]);

        // perform substitution
        retEnc[idx++] = sub(retEnc[idx - 3], i, true);

        // shift rows
        retEnc[idx++] = shiftRows(retEnc[idx - 2], true);

        // mix columns
        retEnc[idx++] = mixColumns(retEnc[idx - 2], i, true);

        // add round key
        retEnc[idx++] = addRound(retEnc[idx - 2], retEnc[idx - 5]);

        }

        // round 10
        retEnc[idx++] = parBlk(rndKeys[10]);

        // perform substitution
        retEnc[idx++] = sub(retEnc[idx - 3], 10, true);

        // shift rows
        retEnc[idx++] = shiftRows(retEnc[idx - 2], true);

        // add round key
        retEnc[idx++] = addRound(retEnc[idx - 2], retEnc[idx - 4]);

        // Set Encrypted Text
        retEnc[idx] = parBlk(retEnc[idx - 1]);

        return retEnc;
}
//##############################################################################################

private int[][] decode(int[] encrypted, int[] cipKey)
{
        int idx = 0;

        // initialize return array
        int[][] retDec = new int[54][];

        for (int i = 0; i < 54; i++)
        {
        retDec[i] = new int[16];
        }

        // record encrypted and cipher key to return array
        retDec[idx++] = parBlk(encrypted);

        retDec[idx++] = cipKey;
```

```
        // round 1
        // record round key
        retDec[idx++] = parBlk(rndKeys[10]);

        // add round key
        retDec[idx++] = addRound(retDec[idx - 2], retDec[idx - 4]);

        // inverse shift rows
        retDec[idx++] = shiftRows(retDec[idx - 2], false);

        // inverse Substitution
        retDec[idx++] = sub(retDec[idx - 2], 10, false);

        for(int i = 9; i > 0; i--)
        {
        // record round key
        retDec[idx++] = parBlk(rndKeys[i]);

        // add round key
        retDec[idx++] = addRound(retDec[idx - 2], retDec[idx - 3]);

        // inverse mix columns
        retDec[idx++] = mixColumns(retDec[idx - 2], i, false);

        // inverse shift rows
        retDec[idx++] = shiftRows(retDec[idx - 2], false);

        // inverse substitution
        retDec[idx++] = sub(retDec[idx - 2], i, false);

        }

        // last round
        // record round key
        retDec[idx++] = parBlk(rndKeys[0]);

        // add round key
        retDec[idx++] = addRound(retDec[idx - 2], retDec[idx - 3]);

        // record decrypted text
        retDec[idx] = parBlk(retDec[idx - 1]);

        return retDec;
}

private void writeResults(int[][] results, bool enc)
{
        if (enc)
        {
        StringBuilder encString = new StringBuilder();

        // write plain text
        encString.Append("Plain Text:\t");

        for(int i = 0; i < 16; i++)
        {
                encString.Append(results[0][i].ToString("X2") + " ");
        }

        encString.Append("\r\n\r\n");

        // write cipher key
```

```
encString.Append("Cipher Key:\t");

for (int i = 0; i < 16; i++)
{
        encString.Append(results[1][i].ToString("X2") + " ");
}

encString.Append("\r\n\r\n");

// write intial block

encString.Append("Initial Block:\t");

for (int i = 0; i < 16; i++)
{
        encString.Append(results[2][i].ToString("X2") + " ");
}

encString.Append("\r\n\r\n");

// write initial round key
encString.Append("Initial Round\r\n");
encString.Append("Key: \t\t");

for (int i = 0; i < 16; i++)
{
        encString.Append(results[3][i].ToString("X2") + " ");
}

encString.Append("\r\n");

// write text after first round key
encString.Append("Add round key: \t");

for (int i = 0; i < 16; i++)
{
        encString.Append(results[4][i].ToString("X2") + " ");
}

encString.Append("\r\n\r\n");

for(int i = 0; i < 9; i++)
{
        encString.Append("Round " + (i+1).ToString() + "\r\n");
        //encString.Append((i + 1).ToString());
        //encString.Append += "\r\n";

        //write round key
        encString.Append("Key: \t\t");

        for (int j = 0; j < 16; j++)
        {
        encString.Append(results[5 * i + 5][j].ToString("X2") + " ");
        }

        encString.Append("\r\n");

        // write text after subbytes
        encString.Append("Subbyte: \t\t");

        for (int j = 0; j < 16; j++)
        {
```

```
        encString.Append(results[5 * i + 6][j].ToString("X2") + " ");
        }

        encString.Append("\r\n");

        // write after shift rows
        encString.Append("Shift Rows: \t");

        for (int j = 0; j < 16; j++)
        {
        encString.Append(results[5 * i + 7][j].ToString("X2") + " ");
        }

        encString.Append("\r\n");

        // write after Mix Columns
        encString.Append("Mix Column: \t");

        for (int j = 0; j < 16; j++)
        {
        encString.Append(results[5 * i + 8][j].ToString("X2") + " ");
        }

        encString.Append("\r\n");

        // write after add round key
        encString.Append("Add Round Key: \t");

        for (int j = 0; j < 16; j++)
        {
        encString.Append(results[5 * i + 9][j].ToString("X2") + " ");
        }

        encString.Append("\r\n\r\n");

}

//round 10
encString.Append("Round 10\r\n");

//write round 10 key
encString.Append("Key: \t\t");

for (int j = 0; j < 16; j++)
{
        encString.Append(results[50][j].ToString("X2") + " ");
}

encString.Append("\r\n");

// write text after subbytes
encString.Append("Subbyte: \t\t");

for (int j = 0; j < 16; j++)
{
        encString.Append(results[51][j].ToString("X2") + " ");
}

encString.Append("\r\n");

// write after shift rows
encString.Append("Shift Rows: \t");
```

```
for (int j = 0; j < 16; j++)
{
        encString.Append(results[52][j].ToString("X2") + " ");
}

encString.Append("\r\n");

// write after add round key
encString.Append("Add Round Key: \t");

for (int j = 0; j < 16; j++)
{
        encString.Append(results[53][j].ToString("X2") + " ");
}

encString.Append("\r\n\r\n");

// write after add round key
encString.Append("Cipher Text: \t");

for (int j = 0; j < 16; j++)
{
        encString.Append(results[54][j].ToString("X2") + " ");
}

encSteps.Clear();

encSteps.Text = encString.ToString();


/* old write method
for (int i = 0; i < 54; i++)
{
        encSteps.Text += "IDX " + i.ToString() + ": ";

        for (int j = 0; j < 16; j++)
        {
        encSteps.Text += results[i][j].ToString("X2") + " ";
        }

        encSteps.Text += "\r\n";
}
*/
}
else
{
StringBuilder decString = new StringBuilder();

// write encrypted text
decString.Append("Encrypted Text: \t");

for (int j = 0; j < 16; j++)
{
        decString.Append(results[0][j].ToString("X2") + " ");

        //decString.Append += " ";
}

decString.Append("\r\n\r\n");

// write cipher key
```

```
decString.Append("Cipher Key: \t");

for (int j = 0; j < 16; j++)
{
        decString.Append(results[1][j].ToString("X2") + " ");

        //decString.Append += " ";
}

decString.Append("\r\n\r\n");

// write round 1
decString.Append("Round 1\r\n");

// write round 1 key
decString.Append("Key: \t\t");

for (int j = 0; j < 16; j++)
{
        //decString.Append += results[2][j].ToString("X2");
        decString.Append(results[2][j].ToString("X2") + " ");

        //decString.Append += " ";
}

//decString.Append += "\r\n";
decString.Append("\r\n");

// write after add round key
decString.Append("Add Round Key: \t");

for (int j = 0; j < 16; j++)
{
        decString.Append(results[3][j].ToString("X2") + " ");

        //decString.Append += " ";
}

decString.Append("\r\n");

// write after inverse shift rows
decString.Append("Inverse Shift Rows:\t");

for (int j = 0; j < 16; j++)
{
        decString.Append(results[4][j].ToString("X2") + " ");

        //decString.Append += " ";
}

decString.Append("\r\n");

// write after inverse subbyte
decString.Append("Inverse Subbyte: \t");

for (int j = 0; j < 16; j++)
{
        decString.Append(results[5][j].ToString("X2") + " ");

        //decString.Append += " ";
}
```

```
decString.Append("\r\n\r\n");

// write rounds 2 - 10
for (int i = 0; i < 9;  i++)
{
        // write round
        decString.Append("Round " + (i+2).ToString() + "\r\n");

        // write round key
        // write last round key
        decString.Append("Key: \t\t");

        for (int j = 0; j < 16; j++)
        {
        decString.Append(results[5 * i + 6][j].ToString("X2") + " ");
        }

        decString.Append("\r\n");

        // write after add round key
        decString.Append("Add Round Key: \t");

        for (int j = 0; j < 16; j++)
        {
        decString.Append(results[5 * i + 7][j].ToString("X2") + " ");
        }

        decString.Append("\r\n");

        // write after inverse mix columns
        decString.Append("Inverse Mix Column:\t");

        for (int j = 0; j < 16; j++)
        {
        decString.Append(results[5 * i + 8][j].ToString("X2") + " ");
        }

        decString.Append("\r\n");

        // write after inverse shift rows
        decString.Append("Inverse Shift Rows:\t");

        for (int j = 0; j < 16; j++)
        {
        decString.Append(results[5 * i + 9][j].ToString("X2") + " ");
        }

        decString.Append("\r\n");

        // write after inverse subbyte
        decString.Append("Inverse Subbyte: \t");

        for (int j = 0; j < 16; j++)
        {
        decString.Append(results[5 * i + 10][j].ToString("X2") + " ");
        }

        decString.Append("\r\n\r\n");

}

// write last round
```

```
        decString.Append("Last Round\r\n");

        // write last round key
        decString.Append("Key: \t\t");

        for (int j = 0; j < 16; j++)
        {
                decString.Append(results[51][j].ToString("X2")+ " ") ;
        }

        decString.Append("\r\n");

        // write after add round key
        decString.Append("Add Round Key: \t");

        for (int j = 0; j < 16; j++)
        {
                decString.Append(results[52][j].ToString("X2") + " ");
        }

        decString.Append("\r\n");

        // write decrypted text
        decString.Append("Decrypted Text:\t");

        for (int j = 0; j < 16; j++)
        {
                decString.Append(results[53][j].ToString("X2") + " ");
        }

        decSteps.Clear();

        decSteps.Text = decString.ToString();

        /* old write method
        decSteps.Clear();
        for (int i = 0; i < 54; i++)
        {
                decSteps.Text += "IDX " + i.ToString() + ": ";

                for (int j = 0; j < 16; j++)
                {
                decSteps.Text += results[i][j].ToString("X2") + " ";
                }

                decSteps.Text += "\r\n";
        }
        */

        }
}

//#########################################################################################

private int[][] genMat(int[] vector)
{
        int[][] retM = new int[vector.Length][];

        for (int i = 0; i < retM.Length; i++)
        {
        retM[i] = new int[vector.Length];
        }
```

```
        for (int i = 0; i < retM.Length; i++)
        {
        for (int j = 0; j < vector.Length; j++)
        {
                int idx = (j - i) % vector.Length;
                while (idx < 0)
                {
                idx += vector.Length;
                }
                retM[i][j] = vector[idx];
        }
        }

        return retM;
}

//#############################################################################################
private int[] genRcon()
{
        int[] retArr = new int[10];

        retArr[0] = 1;

        for (int i = 1; i < retArr.Length; i++)
        {
        int temp = retArr[i - 1] << 1;

        if (temp > 255)
        {
                temp ^= 283; //Think about replacing with sequence
        }

        retArr[i] = (byte)temp;

        }

        return retArr;
}

//#############################################################################################
private int[][] roundKeys()
{
        int[][] retRnd = new int[11][];

        int[][] keyExp = keyExpansion(cipKey);

        for (int i = 0; i < 11; i++)
        {
        retRnd[i] = new int[16];

        for(int j = 0; j < 16; j++)
        {
                retRnd[i][j] = keyExp[j / 4 + i * 4][j % 4];
        }
        }

        return retRnd;
}


private int[][] keyExpansion(int[] key)
```

```
{

        // 44 words: Each word is 4 bytes
        int[][] word = new int[44][];

        // set each word to 4 bytes
        for (int i = 0; i < 44; i++)
        {
        word[i] = new int[4];
        }

        // set words 1 through 4
        for (int i = 0; i < 4; i++)
        {
        for (int j = 0; j < 4; j++)
        {
                word[i][j] = key[4 * i + j];
        }
        }

        // perform key expansion
        for (int i = 4; i < 44; i++)
        {
        //temp = w[i-1]
        int[] temp = new int[4];

        for (int j = 0; j < 4; j++)
        {
                temp[j] = word[i - 1][j];
        }

        if (i % 4 == 0)
        {
                //temp = SubWord(RotWord(temp)) + RCON[i/4];

                //rotate bytes in word one to the left,
               //add rocon[i/4], then sub the byte
                temp = subWord(rotWord(temp), i/4);

                temp[0] ^= rcon[(i / 4) - 1];


        }

        for (int j = 0; j < 4; j++)
        {
                word[i][j] = temp[j] ^ word[i - 4][j];
        }

        //word[i] = temp;

        }
        return word;
}
private int[] rotWord(int[] word)
{
        int[] retWord = new int[4];

        for (int i = 0; i < 4; i++)
        {
        retWord[i] = word[(i + 1) % 4];
        }
```

```
        return retWord;
}

private int[] subWord(int[] word, int round)
{

        int[] retWord = new int[4];

        for (int i = 0; i < 4; i++)
        {
        retWord[i] = subLkUp(word[i], sBoxes[rkSeq[round]-1]);
        }

        return retWord;
}

private int genSubByte(int poly, int mod)
{

        //find inverse of current number
        string polyInv = writePoly(findInv(poly, mod));

        // set inverse vecter
        int[] invVector = new int[8];

        for (int i = 0; i < polyInv.Length; i++)
        {
        invVector[polyInv.Length - i - 1] = int.Parse(polyInv[i].ToString());
        }

        // multiply inverse vector and matrix and add offset vector
        int[] invVectorM = mvmul(baseMatrix, invVector);

        for (int i = 0; i < invVectorM.Length; i++)
        {
        invVectorM[i] = (invVectorM[i] + baseOffV[i]) % 2;
        }


        return vToI(invVectorM);

}

private int genInvSubByte(int poly, int mod)
{
        string polyString = writePoly(poly);

        int[] invVector = new int[8];

        for (int i = 0; i < polyString.Length; i++)
        {
        invVector[polyString.Length - i - 1] = int.Parse(polyString[i].ToString());
        }

        // good to here
        int[] invVectorM = mvmul(invMatrix, invVector);

        for (int i = 0; i < invVectorM.Length; i++)
        {
        invVectorM[i] = (invVectorM[i] + invOffV[i]) % 2;
        }
```

```
        int invPoly = vToI(invVectorM);

        return findInv(invPoly, mod);

}

private String writePoly(int poly)
{
        StringBuilder sb = new StringBuilder();
        while (poly > 0)
        {
        sb.Insert(0, poly % 2);
        poly /= 2;
        }

        while (sb.Length < 8)
        {
        sb.Insert(0, 0);
        }

        return sb.ToString();
}

private int vToI(int[] arr)
{
        int val = 0;

        for (int i = 0; i < arr.Length; i++)
        {
        if (arr[i] == 1)
        {
                val += (int)Math.Pow(2, i);
        }
        }

        return val;
}

private int[] mvmul(int[][] A, int[] b)
{
        int rowA = A.Length;
        int colA = A[0].Length;

        int rowB = b.Length;

        int[] retM = new int[rowA];

        for (int i = 0; i < rowA; i++)
        {
        for (int j = 0; j < colA; j++)
        {
                retM[i] += A[i][j] * b[j];
        }
        }

        return retM;
}

private int findInv(int poly, int mod)
{
        if (poly == 0 || poly == 1)
```

```
                {
                return poly;
                }

                int[] initQR = divide(mod, poly);

                int[][] invArr = new int[3][];

                for (int i = 0; i < invArr.Length; i++)
                {
                invArr[i] = new int[3];
                }

                // Set initial values
                invArr[0][0] = mod;
                invArr[0][1] = 1;
                invArr[0][2] = 0;

                invArr[1][0] = poly;
                invArr[1][1] = 0;
                invArr[1][2] = 1;

                int idx = 2;


                while (true)
                {
                // calculate quotient and remainder
                int[] qr = divide(invArr[(idx - 2) % 3][0], invArr[(idx - 1) % 3][0]);
                invArr[idx % 3][0] = qr[1];

                // calculate new u
                invArr[idx % 3][1] = invArr[(idx - 2) % 3][1] ^
                                     polyMult(invArr[(idx - 1) % 3][1], qr[0]);

                // calculate new v
                invArr[idx % 3][2] = invArr[(idx - 2) % 3][2] ^ polyMult(invArr[(idx - 1) % 3][2],
qr[0]);

                int[] newQR = divide(qr[0], invArr[idx % 3][1]);

                if (invArr[idx % 3][0] == 1)
                {
                        return invArr[idx % 3][2];
                }

                idx++;

                }

        }

        private int[] divide(int poly, int div)
        {
                int quotient = 0;
                int remainder = poly;
                //int temp = div;

                int resultSize = bitCount(poly);
                int divSize = bitCount(div);

                while (resultSize >= divSize)
```

```
        {
        int tempQuo = 1 << (resultSize - divSize);

        quotient ^= tempQuo;

        int temp = div << (resultSize - divSize);

        remainder = remainder ^ temp;

        resultSize = bitCount(remainder);
        }

        int[] retArr = { quotient, remainder };

        return retArr;
}

private int polyMult(int a, int b)
{
        int product = 0;

        string aBits = writePoly(a);
        string bBits = writePoly(b);

        for (int i = 0; i < 8; i++)
        {
        if (aBits[i] == '1')
        {
                int aVal = 1 << aBits.Length - i - 1;
                int aTemp = 0;

                for (int j = 0; j < bBits.Length; j++)
                {
                if (bBits[j] == '1')
                {
                        int bTemp = aVal << bBits.Length - j - 1;
                        aTemp = aTemp ^ bTemp;
                }
                }

                product = product ^ aTemp;

        }
        }

        return product;
}

private int bitCount(int x)
{
        return (int)Math.Log((double)x, 2.0) + 1;
}

private int bToI(int[] arr)
{
        int val = 0;

        for (int i = 0; i < arr.Length; i++)
        {
        if (arr[i] == 1)
        {
                val += (int)Math.Pow(2, arr.Length - i - 1);
```

```
                }
                }
                return val;
        }

        //################################################################################################

        private int[] addRound(int[] rndKey, int[] plnTxt)
        {
                int[] retText = new int[16];

                for (int i = 0; i < 16; i++)
                {
                retText[i] = rndKey[i] ^ plnTxt[i];
                }

                return retText;

        }

        //################################################################################################

        private int subLkUp(int unsubbed, int[] sBox)
        {
                int left = unsubbed >> 4;
                int right = unsubbed ^ (left << 4);
                return sBox[left * 16 + right];
        }


        //################################################################################################

        private int[] sub(int[] unsubbed, int round, bool enc)
        {
                int[] subbed = new int[16];

                for (int i = 0; i < 16; i++)
                {
                //subbed[i] = (byte)genSubByte(unsubbed[i], polyList[sbSeq[round]]);
                if (enc)
                {
                        subbed[i] = subLkUp(unsubbed[i], sBoxes[sbSeq[round]-
1]);//(byte)genSubByte(unsubbed[i], polyList[sbSeq[round]]);
                }
                else
                {
                        subbed[i] = subLkUp(unsubbed[i], isBoxes[sbSeq[round]-
1]);//(byte)genInvSubByte(unsubbed[i], polyList[sbSeq[round]]);
                }
                }

                return subbed;
        }

        //################################################################################################

        private int[] shiftRows(int[] unshifted, bool enc)
        {
                int[] shifted = new int[16];

                for (int i = 0; i < 4; i++)
                {
```

```
        for (int j = 0; j < 4; j++)
        {
                if (enc)
                {
                shifted[i * 4 + j] = unshifted[(i * 4) + ((j + i) % 4)];
                }
                else
                {
                shifted[i * 4 + j] = unshifted[(i * 4) + ((j + 4 - i) % 4)];
                }
        }
        }

        return shifted;
}

//#############################################################################################

private int[] mixColumns(int[] shifted, int round, bool enc)
{

        int[][] mixedMatrix = new int[4][];

        for (int i = 0; i < 4; i++)
        {
        mixedMatrix[i] = new int[4];

        for (int j = 0; j < 4; j++)
        {
                mixedMatrix[i][j] = shifted[i * 4 + j];
                if(round == 1)
                {

                }
        }

        }


        if (enc)
        {
        mixedMatrix = mmpMul(mcMatrix, mixedMatrix, round);
        }
        else
        {
        mixedMatrix = mmpMul(imcMatrix, mixedMatrix, round);
        }

        int[] retVector = new int[16];

        for (int i = 0; i < 16; i++)
        {
        retVector[i] = mixedMatrix[i / 4][i % 4];
        }

        return retVector;
}

private int[][] mmpMul(int[][] a, int[][] b, int round)
{
        int[][] retMat = new int[4][];
```

```
for (int i = 0; i < 4; i++)
{
retMat[i] = new int[4];
}

for (int i = 0; i < 4; i++)
{
for (int j = 0; j < 4; j++)
{
        int sum = 0;

        for (int k = 0; k < 4; k++)
        {
        int tempVal = b[k][j];
        int tempVal2 = b[k][j];
        int tempVal3 = b[k][j];

        switch (a[i][k])
        {
                case 1:
                sum ^= tempVal;
                break;

                case 2:

                tempVal <<= 1;


                if (tempVal > 255)
                {
                        tempVal ^= polyList[sbSeq[round]];  //283;
                }

                sum ^= tempVal;

                break;

                case 3:

                tempVal <<= 1;

                if (tempVal > 255)
                {
                        tempVal ^= polyList[sbSeq[round]]; // 283;
                }

                sum ^= tempVal ^ b[k][j];

                break;

                case 9:

                for (int l = 0; l < 3; l++)
                {
                        tempVal <<= 1;

                        if (tempVal > 255)
                        {
                        tempVal ^= polyList[sbSeq[round]]; // 283;
                        }
                }
```

```
sum ^= tempVal ^ b[k][j];

break;

case 11:

for (int l = 0; l < 3; l++)
{
        tempVal3 <<= 1;

        if (tempVal3 > 255)
        {
        tempVal3 ^= polyList[sbSeq[round]]; // 283;
        }
}

tempVal <<= 1;

if (tempVal > 255)
{
        tempVal ^= polyList[sbSeq[round]]; //283;
}

sum ^= tempVal3 ^ tempVal ^ b[k][j];


break;

case 13:

for (int l = 0; l < 3; l++)
{
        tempVal3 <<= 1;

        if (tempVal3 > 255)
        {
        tempVal3 ^= polyList[sbSeq[round]]; //283;
        }
}

for (int l = 0; l < 2; l++)
{
        tempVal2 <<= 1;

        if (tempVal2 > 255)
        {
        tempVal2 ^= polyList[sbSeq[round]]; //283;
        }
}

sum ^= tempVal3 ^ tempVal2 ^ b[k][j];

break;

case 14:

for (int l = 0; l < 3; l++)
{
        tempVal3 <<= 1;

        if (tempVal3 > 255)
        {
```

```
                        tempVal3 ^= polyList[sbSeq[round]]; //283;
                        }
                }

                for (int l = 0; l < 2; l++)
                {
                        tempVal2 <<= 1;

                        if (tempVal2 > 255)
                        {
                        tempVal2 ^= polyList[sbSeq[round]]; // 283;
                        }
                }

                tempVal <<= 1;

                if (tempVal > 255)
                {
                        tempVal ^= polyList[sbSeq[round]]; // 283;
                }


                sum ^= tempVal3 ^ tempVal2 ^ tempVal;

                break;

        }

        }

        if (sum > 255)
        {
        retMat[i][j] = (byte)(sum ^ polyList[sbSeq[round]]); // 283);
        }
        else
        {
        retMat[i][j] = (byte)sum;
        }

    }
    }

    return retMat;

}

private void button1_Click(object sender, EventArgs e)
{

}

int[] parBlk(int[] block)
{
    int[,] temp = new int[4, 4];
    int[] temp2 = new int[16];
    for(int i = 0; i < 4; i++)
    {
    for(int j=0; j < 4; j++)
    {
            temp[i, j] = block[i * 4 + j];
    }
    }
```

```
                for(int i=0; i<16; i++)
                {
                temp2[i] = temp[i % 4, i / 4];

                }

                return temp2;
        }
    }
}
```

Appendix C: S-Box Verification Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Collections;

namespace S_Box_Verification
{
        public partial class Form1 : Form
        {

        ArrayList divisors = new ArrayList();
        ArrayList irPolys = new ArrayList();
        bool irreducible;
        int degree = 8;

        long[][] baseMatrix;
        long[][] invMatrix;

        long[] baseV = { 1, 0, 0, 0, 1, 1, 1, 1 };
        long[] invV = { 0, 0, 1, 0, 0, 1, 0, 1 };

        long[] baseOffV = { 1, 1, 0, 0, 0, 1, 1, 0 };
        long[] invOffV = { 1, 0, 1, 0, 0, 0, 0, 0 };

        long[] result = new long[256];

        string[] super =
{
                "1",
                "x",
                "x\u00b2",
                "x\u00b3",
                "x\u2074",
                "x\u2075",
                "x\u2076",
                "x\u2077",
                "x\u2078",
                "x\u2079",
                "x\u00b9" + "\u2070",
                "x\u00b9" + "\u00b9",
                "x\u00b9" + "\u00b2",
                "x\u00b9" + "\u00b3",
                "x\u00b9" + "\u2074",
                "x\u00b9" + "\u2075",
                "x\u00b9" + "\u2076",
        };


        public Form1()
        {
                InitializeComponent();
```

```
        //statLbl.Text = "Status: Running";
        //statLbl.Refresh();

        //clear previous information
        irPolys.Clear();
        divisors.Clear();
        irPolyList.Items.Clear();

        //irPolyCntLbl.Text = "Total: 0";
        //prPolyCntLbl.Text = "Total: 0";

        //Get Degree
        //int.TryParse(degIn.Text, out degree);

        if (degree < 1 || 16 < degree)
        {
        return;
        }


        //Compute Lower degree irreducible polynomials

        for (int d = 1; d <= (int)degree / 2; d++)
        {
        long[][] curDeg = generate((long)d);

        for (int i = 0; i < curDeg.Length; i++)
        {
                irreducible = true;

                long curPoly = bToI(curDeg[i]);

                foreach (long div in divisors)
                {
                if (divide(curPoly, div)[1] == 0)
                {
                        irreducible = false;
                        break;
                }
                }

                if (irreducible)
                {
                divisors.Add(curPoly);
                }
        }
        }

        //Generate irreducible polynomials
        long[][] irGen = generate(degree);
        reduce(irGen);

        foreach (long poly in irPolys)
        {

        irPolyList.Items.Add(outPoly(writePoly(poly)));
        }

        baseMatrix = genMat(baseV);

        invMatrix = genMat(invV);
}
```

```
        //##############################//
        // BUTTONS                      //
        //##############################//
        private void VerBtn_Click(object sender, EventArgs e)
        {
                if (irPolyList.SelectedItem != null)
                {
                int[] sBoxDups = new int[256];
                int[] isBoxDups = new int[256];
                int[] sMapCheck = new int[256];
                int[] isMapCheck = new int[256];

                string modStr =
writePoly(long.Parse(irPolys[irPolyList.SelectedIndex].ToString()));


                int mod = (int)sToI(modStr);


                for (int i = 0; i < 256; i++)
                {

                        long sBoxVal = genSubByte(i, mod);
                        long isBoxVal = genInvSubByte(i, mod);

                        sBoxList.Items.Add(sBoxVal.ToString("X2"));
                        isBoxList.Items.Add(isBoxVal.ToString("X2"));

                        stimList.Items.Add(i.ToString("X2") + " : " +
findMap(parseHex(sBoxList.Items[i].ToString())).ToString("X2"));
                        itsmList.Items.Add(i.ToString("X2") + " : " +
findMap(parseHex(isBoxList.Items[i].ToString())).ToString("X2"));

                        sBoxDups[sBoxVal] += 1;
                        isBoxDups[isBoxVal] += 1;

                        sMapCheck[i] = (int)sBoxVal;
                        isMapCheck[i] = (int)isBoxVal;


                }

                //perform checks
                for (int i = 0; i < 256; i++)
                {
                        if (sBoxDups[i] != 1)
                        {
                        sBoxDupsList.Items.Add(i);
                        }

                        if (isBoxDups[i] != 1)
                        {
                        isBoxDupsList.Items.Add(i);
                        }

                        if(isMapCheck[sMapCheck[i]] != i)
                        {
                        mapCheckList.Items.Add(i);
                        }
                }
                }
```

```
}

//##############################//
// PRIVATE METHODS              //
//##############################//
private int bitCount(long x)
{
        return (int)Math.Log((double)x, 2.0) + 1;
}

private long bToI(long[] arr)
{
        long val = 0;

        for (int i = 0; i < arr.Length; i++)
        {
        if (arr[i] == 1)
        {
                val += (long)Math.Pow(2, arr.Length - i - 1);
        }
        }
        return val;
}

private long[] divide(long poly, long div)
{
        long quotient = 0;
        long remainder = poly;
        long temp = div;

        int resultSize = bitCount(poly);
        int divSize = bitCount(div);

        while (resultSize >= divSize)
        {
        long tempQuo = 1 << resultSize - divSize;

        quotient ^= tempQuo;

        temp = div << resultSize - divSize;

        remainder = remainder ^ temp;

        resultSize = bitCount(remainder);
        }

        long[] retArr = { quotient, remainder };

        return retArr;
}

private long[] findGCD(long poly, long mod)
{
        return divide(mod, poly);
}

private long findInv(int poly, int mod)
{
        if (poly == 0 || poly == 1)
        {
        return poly;
        }
```

```
        long[] initQR = findGCD(poly, mod);

        long[][] invArr = new long[3][];

        for (int i = 0; i < invArr.Length; i++)
        {
        invArr[i] = new long[3];
        }

        // Set initial values
        invArr[0][0] = mod;
        invArr[0][1] = 1;
        invArr[0][2] = 0;

        invArr[1][0] = poly;
        invArr[1][1] = 0;
        invArr[1][2] = 1;

        int idx = 2;


        while (true)
        {
        // calculate quotient and remainder
        long[] qr = divide(invArr[(idx - 2) % 3][0], invArr[(idx - 1) % 3][0]);
        invArr[idx % 3][0] = qr[1];

        // calculate new u
        invArr[idx%3][1] = invArr[(idx-2)%3][1]^polyMult(invArr[(idx-1)%3][1],qr[0]);

        // calculate new v
        invArr[idx%3][2]=invArr[(idx-2)%3][2]^polyMult(invArr[(idx-1)%3][2],qr[0]);

        long[] newQR = divide(qr[0], invArr[idx % 3][1]);

        if (invArr[idx % 3][0] == 1)
        {
                return invArr[idx % 3][2];
        }

        idx++;

        }

}

private int findMap(int[] hb)
{
        return 16 * hb[0] + hb[1];
}

private long[][] generate(long degree)
{
        // create array to house polynomials
        long[][] gen = new long[(int)(Math.Pow(2, (double)degree + 1) / 2)][];

        for (int i = 0; i < gen.Length; i++)
        {
        gen[i] = new long[(int)degree + 1];
        }
```

```
        // generatate all polynomials of given degree
        for (int i = 0; i < gen.Length; i++)
        {
        //must have a 1 in the first term to be a polynomial of degree
        gen[i][0] = 1;

        // generate all forms of lower degrees
        for (int j = 1; j < gen[i].Length; j++)
        {
                gen[i][j] = (long)(i / Math.Pow(2, gen[i].Length - j - 1)) % 2;
        }
        }

        return gen;
}

private long genInvSubByte(int poly, int mod)
{
        string polyString = writePoly(poly);

        long[] invVector = new long[8];

        for (int i = 0; i < polyString.Length; i++)
        {
        invVector[polyString.Length - i - 1] = (long)int.Parse(polyString[i].ToString());
        }

        // good to here
        long[] invVectorM = mvmul(invMatrix, invVector);

        for (int i = 0; i < invVectorM.Length; i++)
        {
        invVectorM[i] = (invVectorM[i] + invOffV[i]) % 2;
        }

        int invPoly = (int)vToI(invVectorM);

        return findInv(invPoly, mod);

}

private long[][] genMat(long[] vector)
{
        long[][] retM = new long[vector.Length][];

        for (int i = 0; i < retM.Length; i++)
        {
        retM[i] = new long[vector.Length];
        }

        for (int i = 0; i < retM.Length; i++)
        {
        for (int j = 0; j < vector.Length; j++)
        {
                int idx = (j - i) % vector.Length;
                while (idx < 0)
                {
                idx += vector.Length;
                }
                retM[i][j] = vector[idx];
        }
        }
```

```
        return retM;
}

private long genSubByte(int poly, int mod)
{

        //find inverse of current number
        string polyInv = writePoly(findInv(poly, mod));

        // set inverse vecter
        long[] invVector = new long[8];

        for (int i = 0; i < polyInv.Length; i++)
        {
        invVector[polyInv.Length - i - 1] = (long)int.Parse(polyInv[i].ToString());
        }

        // multiply inverse vector and matrix and add offset vector
        long[] invVectorM = mvmul(baseMatrix, invVector);

        for (int i = 0; i < invVectorM.Length; i++)
        {
        invVectorM[i] = (invVectorM[i] + baseOffV[i]) % 2;
        }


        return vToI(invVectorM);

}

private long[] mvmul(long[][] A, long[] b)
{
        int rowA = A.Length;
        int colA = A[0].Length;

        int rowB = b.Length;

        long[] retM = new long[rowA];

        if (colA != rowB)
        {
        Console.WriteLine("Dimensional Mismatch");
        }
        else
        {
        for (int i = 0; i < rowA; i++)
        {
                for (int j = 0; j < colA; j++)
                {
                retM[i] += A[i][j] * b[j];
                }
        }
        }

        return retM;
}

private string outPoly(string write)
{

        StringBuilder sb2 = new StringBuilder();
```

```
        for (int i = 0; i < write.Length; i++)
        {
        if (write[i] == '1')
        {
                sb2.Append(super[write.Length - i - 1]);

                if (i < write.Length - 1)
                {
                sb2.Append(" + ");
                }
        }

        }

        return sb2.ToString();
}

private long polyMult(long a, long b)
{
        long c = 0;

        string aBits = writePoly(a);
        string bBits = writePoly(b);

        for (int i = 0; i < aBits.Length; i++)
        {
        if (aBits[i] == '1')
        {
                long aVal = 1 << aBits.Length - i - 1;
                long aTemp = 0;

                for (int j = 0; j < bBits.Length; j++)
                {
                if (bBits[j] == '1')
                {
                        long bTemp = aVal << bBits.Length - j - 1;
                        aTemp = aTemp ^ bTemp;
                }
                }

                c = c ^ aTemp;

        }
        }

        return c;
}

private int[] parseHex(string hb)
{
        int[] retArr = new int[2];

        for(int i = 0; i < hb.Length; i++)
        {
        if((char)hb[i] >= 'A' && (char)hb[i] <= 'F')
        {
                retArr[i] = hb[i] - 'A' + 10;
        }

        else
        {
                int.TryParse(hb[i].ToString(), out retArr[i]);
```

```
                }
                }

                return retArr;
        }

        private void reduce(long[][] arr)
        {
                long count = 0;

                for (int i = 0; i < arr.Length; i++)
                {
                long curPoly = bToI(arr[i]);

                // if the number of terms is even, the polynomial is reducible

                long sum = 0;

                for (int j = 0; j < arr[i].Length; j++)
                {
                        sum += arr[i][j];
                }

                if (sum % 2 == 0)
                {
                        continue;
                }

                // if the coefficient of last term is 0 then we can always factor a first degree
polynomial
                if (arr[i][arr[i].Length - 1] == 0)
                {
                        continue;
                }

                // otherwise potentially irreducible must check
                irreducible = true;

                foreach (long div in divisors)
                {
                        if (divide(curPoly, div)[1] == 0)
                        {
                        irreducible = false;
                        break;
                        }
                }

                if (irreducible)
                {
                        irPolys.Add(curPoly);
                        count++;
                }

                }
        }

        private long sToI(string s)
        {
                long[] arr = new long[s.Length];

                for (int i = 0; i < s.Length; i++)
                {
```

```
                arr[i] = long.Parse(s[i].ToString());
                }

                return bToI(arr);
        }

        private long vToI(long[] arr)
        {
                long val = 0;

                for (int i = 0; i < arr.Length; i++)
                {
                if (arr[i] == 1)
                {
                        val += (long)Math.Pow(2, i);
                }
                }

                return val;
        }

        private String writePoly(long poly)
        {
                StringBuilder sb = new StringBuilder();
                while (poly > 0)
                {
                sb.Insert(0, poly % 2);
                poly /= 2;
                }

                return sb.ToString();
        }
        }
}
```

References

[1] Aumasson, J. P. (2017). The impact of quantum computing on cryptography. *Computer Fraud & Security*, *2017*(6), 8-11.

[2] Mosca, M. (2018). Cybersecurity in an era with quantum computers: will we be ready?. *IEEE Security & Privacy*, *16*(5), 38-41.

[3] Chen, L., Chen, L., Jordan, S., Liu, Y. K., Moody, D., Peralta, R., ... & Smith-Tone, D. (2016). *Report on post-quantum cryptography* (Vol. 12). US Department of Commerce, National Institute of Standards and Technology.

[4] Mavroeidis, V., Vishi, K., Zych, M. D., & Jøsang, A. (2018). The impact of quantum computing on present cryptography. *arXiv preprint arXiv:1804.00200*.

[5] Bernstein, D. J., & Lange, T. (2017). Post-quantum cryptography. *Nature*, *549*(7671), 188-194.

[6] Singh, G. (2013). A study of encryption algorithms (RSA, DES, 3DES and AES) for information security. *International Journal of Computer Applications*, *67*(19).

[7] Yfantis, E. A. (2018, January). A new cyclic cryptographic algorithm. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)* (pp. 20-25). IEEE.

[8] Grover, L. K. (1996, July). A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (pp. 212-219).

[9] Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, *41*(2), 303-332.

[10] Zalka, C. (1999). Grover's quantum searching algorithm is optimal. *Physical Review A*, *60*(4), 2746.

[11] Grassl, M., Langenberg, B., Roetteler, M., & Steinwandt, R. (2016, February). Applying Grover's algorithm to AES: quantum resource estimates. In *Post-Quantum Cryptography* (pp. 29-43). Springer, Cham.

[12] Daemen, J., & Rijmen, V. (1999). AES proposal: Rijndael.

[13] National Institute of Standards and Technology (2001). Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, *197*(1-51), 3-3.https://doi.org/10.6028/NIST.FIPS.197

[14] Heron, S. (2009). Advanced encryption standard (AES). *Network Security*, *2009*(12), 8-12.

[15] Daemen, J., & Rijmen, V. (2010). The first 10 years of advanced encryption. *IEEE Security & Privacy*, *8*(6), 72-74.

[16] Dragomir, I. R., & Lazăr, M. (2016, June). Generating and testing the components of a block cipher. In *2016 8th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)* (pp. 1-4). IEEE.

[17] Dragomir, I. R., Măluţan, S. B., & Lazăr, M. (2019, June). An analysis of cryptographic algorithm strength based on S-box properties. In *2019 11th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)* (pp. 1-4). IEEE.

[18] Mohamed, K., Pauzi, M. N. M., Ali, F. H. H. M., Ariffin, S., & Zulkipli, N. H. N. (2014, September). Study of S-box properties in block cipher. In *2014 International Conference on Computer, Communications, and Control Technology (I4CT)* (pp. 362-366). IEEE.

[19] Xian, Z. H., & Sun, S. L. (2010, March). Study on test for structure of S-Boxes in Rijndael. In *2010 Second International Workshop on Education Technology and Computer Science* (Vol. 3, pp. 84-86). IEEE.

[20] Das, S., Zaman, J. U., & Ghosh, R. (2013). Generation of AES S-Boxes with various modulus and additive constant polynomials and testing their randomization. *Procedia Technology*, *10*, 957-962.

[21] Nyberg, K. (1991, April). Perfect nonlinear S-boxes. In *Workshop on the Theory and Application of Cryptographic Techniques* (pp. 378-386). Springer, Berlin, Heidelberg.

[22] Radhakrishnan, S. V., & Subramanian, S. (2013). An analytical approach to s-box generation. *Computers & Electrical Engineering*, *39*(3), 1006-1015.

[23] Khamlich, E. (2013). Implementation of stronger AES by using Dynamic S-Box dependent of Master Key. *Journal of Theoretical and Applied Information Technology*, *53*(2).

[24] Hosseinkhani, R., & Javadi, H. H. S. (2012). Using cipher key to generate dynamic S-box in AES cipher system. *International Journal of Computer Science and Security (IJCSS)*, *6*(1), 19-28.

[25] El-Ramly, S. H., El-Garf, T., & Soliman, A. H. (2001, March). Dynamic generation of S-boxes in block cipher systems. In *Proceedings of the Eighteenth National Radio Science Conference. NRSC'2001 (IEEE Cat. No. 01EX462)* (Vol. 2, pp. 389-397). IEEE.

[26] Karki, P., Das, S. R., Biswas, S. N., Assaf, M. H., Groza, V., Petriu, E. M., & Morton, S. (2015). Advanced encryption techniques based on S-box and elliptical curve revisited. *SDPS-2015, Society for Design and Process Science*.

[27] Juremi, J., Mahmod, R., & Sulaiman, S. (2012, June). A proposal for improving AES S-box with rotation and key-dependent. In *Proceedings Title: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec)* (pp. 38-42). IEEE.

[28] Janadi, A., & Tarah, D. A. (2008, April). AES immunity Enhancement against algebraic attacks by using dynamic S-Boxes. In *2008 3rd International Conference on Information and Communication Technologies: From Theory to Applications* (pp. 1-6). IEEE.

[29] Bassham III, L. E. (2002). The advanced encryption standard algorithm validation suite (AESAVS). *NIST Information Technology Laboratory*.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Ashby Mullin
amullin20@gmail.com

Degrees:

Bachelor of Science in Computer Science 2018

Doane University

Thesis Title: A fortified extension of the AES and its Implementation

Thesis Examination Committee:

Chairperson, Dr. Evangelos Yfantis, Ph.D.

Committee Member, Dr. Hal Berghel, Ph.D.

Committee Member, Dr. Andreas Stefik, Ph.D.

Graduate Faculty Representative, Dr. Sarah Harris, Ph.D.