9-2007

Cellular automata for dynamic S-boxes in cryptography.

William Matthew Luckett
University of Louisville

Recommended Citation

Luckett, William Matthew, "Cellular automata for dynamic S-boxes in cryptography." (2007). *Electronic Theses and Dissertations.* Paper 863.
https://doi.org/10.18297/etd/863

CELLULAR AUTOMATA FOR DYNAMIC S-BOXES IN CRYPTOGRAPHY

By

William Matthew Luckett
B.S., University of Louisville, 2005

A Thesis
Submitted to the Faculty of the
University of Louisville
J. B. Speed School of Engineering
in Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Computer Engineering and Computer Science

September 2007

CELLULAR AUTOMATA FOR DYNAMIC S-BOXES IN CRYPTOGRAPHY

Submitted by: _____

William Luckett

A Thesis Approved On

_____

(Date)

by the Following Reading and Examination Committee

_____

Dr. Mehmed Kantardzic, Thesis Director

_____

Dr. Ahmed Desoky

_____

Dr. John Naber

ABSTRACT

In today's world of private information and mass communication, there is an ever increasing need for new methods of maintaining and protecting privacy and integrity of information. This thesis attempts to combine the chaotic world of cellular automata and the paranoid world of cryptography to enhance the S-box of many Substitution Permutation Network (SPN) ciphers, specifically Rijndael/AES. The success of this enhancement is measured in terms of security and performance.

The results show that it is possible to use Cellular Automata (CA) to enhance the security of an 8-bit S-box by further randomizing the structure. This secure use of CA to scramble the S-box, removes the "9-term algebraic expression" [20] [21] that typical Galois generated S-boxes share. This cryptosystem securely uses a Margolis class, partitioned block, uniform gas, cellular automata to create unique S-boxes for each block of data to be processed.

The system improves the base Rijndael algorithm in the following ways. First, it utilizes a new S-box for each block of data. This effectively limits the amount of data that can be gathered for statistical analysis to the blocksize being used. Secondly, the S-boxes are not stored in the compiled binary, which protects against an "S-box blanking" [22] attack. Thirdly, the algebraic expression hidden within each galois generated S-box is destroyed after one CA generation, which also modifies key expansion results. Finally, the thesis succeeds in combining Cellular Automata and Cryptography securely, though it is not the most efficient solution to dynamic S-boxes.

TABLE OF CONTENTS

INDEX OF APPENDICES

INDEX OF TABLES

INDEX OF FIGURES

1. INTRODUCTION

Since their conception by Stanislaw Ulam and John von Neumann in the 1940's [1], researchers have successfully utilized Cellular Automata (CA) for many different purposes. The simple structure and the properties of CA suggest various uses in many fields of study. However, it has proven quite difficult to integrate CA with the field of cryptography. Despite previous results, new uses for CA in cryptographic applications are often investigated. This thesis will attempt to enhance the security of the Rijndael encryption algorithm by using the perceived performance and randomness of CA to modify the typically static S-Box structure. While the strength of the existing Rijndael algorithm may not reside entirely with the S-Box, research suggests room for improvement in both implementation and the design of S-Box dependent algorithms.

The thesis will consist of three main chapters each having several sub chapters. The introduction will provide relevant background information over the topics of Cellular Automata and cryptography. The typical operation of common types of CA will be explained, as well as their perceived uses in cryptography. Similarly, common cryptographic networks and primitives, like the S-Box, will be introduced. This first chapter will finally provide a look at some previously proposed Cellular Automata Cryptosystems (CACs) and their shortcomings.

The second chapter of the thesis will focus on the design and implementation of the proposed system. The proposal integrates two separate systems into one and was

designed similarly. In the first part of the second chapter, the Rijndael algorithm is explained. This section details the structures and operations that work together to make Rijndael a secure algorithm. The chapter then explains the S-Box modifying CA. The design and operation of the CA as well as its properties and selection criteria are rationalized. The chapter finally shows the integration of the two parts, as the proposal suggests.

The proposed CAC is analyzed in the third chapter based on a number of criteria. The two halves of the CAC, the cipher and the CA, are analyzed independently to verify their operation and fulfillment of their selection criteria. Finally, the combined system is evaluated based on strength of the generated S-Boxes, the security of the implementation, and the processing performance. The final results of the analysis lead to the conclusion that the CAC operates correctly, offers improvements to Rijndael's S-Box, and has a moderate impact on performance.

1.1. Cellular Automata

Cellular Automata typically exist on a finite or infinite regular grid of cells. Each cell, or automata, has a finite number of possible states. These automata cells are each modified independently by the transition function on a discrete time step. The application of the transition function to each cell in the grid leads to the next 'generation' for the grid. Transition functions typically operate based on the state of the cell and the cells around it. Though these functions depend on their input states, every cell follows the same rule for determining these transitions. Formally, a CA is a 4-tuple [1] [2]:

CA = (d, S, N, _f_)
where **d** = is a triple determining the size of dimensions
S = the finite set of possible states
N = the neighborhood vector for each cell
_f_ = the local transition function

1.1.1. CA Dimensions.

The dimensions defined by **d** could be described as $N_x \times N_y \times N_z$, where (N_x, N_y, N_z) are members of the natural numbers. Typically CA are limited to one or two dimensions $(N_{z\,=\,0})$ and a finite size for the dimensions N_x and N_y. In an attempt to emulate an infinite grid, the border cells typically have their neighbors wrap to the other side of the grid, creating a N_z-dimensional torus [2], as shown in the image [3] in Figure 1.
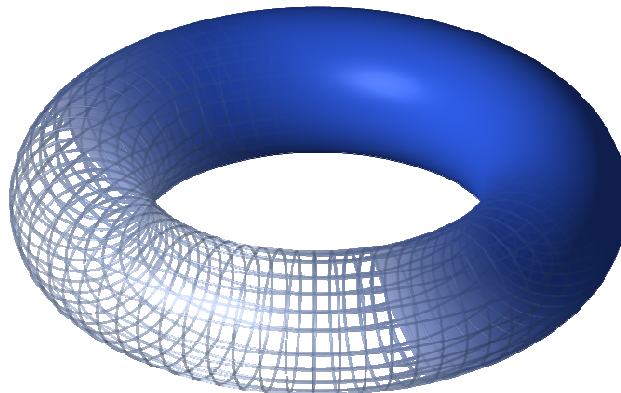
Figure 1. Toroidal result of wrapping borders of a 2-D plane. [3]

1.1.2. CA States.

The finite set of states **S** = $(S_0, S_1, \ldots, S_{n-1})$ where n is the size of the set **S**. Most CA based on Margolus's Lattice Gas Automata or Conway's Game of Life will have only two states. For Lattice Gas Automata, the two states are seen as the existence or non-existence of a particle. In Conway's Game of Life, the two states indicate whether a cell is *alive* or *dead*. Few automata have more than two states, as the design complexity

of the transition function f increases with the number of states. The transition function f transforms a cell's state to another state based on its current state and its neighbor's states defined by **N**. The combined states of the cells in the neighborhood is called the *configuration*. The number of possible configurations for a neighborhood depends on the size of the state set **S** and the number of cells in neighborhood **N**, which amounts to $\mathbf{S^N}$ total configurations for a specific CA.

1.1.3. CA Neighborhoods.

For the standard 2-dimensional cellular automaton on a square grid, there are three standard choices of neighborhood vector **N**. Edward F. Moore proposed one method [4] for defining neighborhoods. A Moore neighborhood of range r is defined by $N_{Moore}(x_0, y_{0,r}) = [(x, y) : |x - x_0| \le r, |y - y_0| \le r]$, and the number of cells in each neighborhood is $(2r + 1)^2$. A r value of one is typically used which yields eight neighbors to each cell, for a total of nine cells in the Moore neighborhood as seen in Figure 2a. The Moore neighbor-hooding scheme is used in John Conway's Game of Life, and many related CA [2]. The second most common scheme is the von Neumann neighborhood [5]. A von Neumann neighborhood of range r is defined by $N_{vonNeumann}(x_0, y_{0,r}) = [(x, y) : |x - x_0| + |y - y_0| \le r]$, and the number of cells in each neighborhood is $2r(r + 1) + 1$. A range of one is typically used which yields four neighbors to each cell, for a total of five cells in the von Neumann neighborhood, as seen in Figure 2b.

x_{x-1},y_{y-1}	x_x,y_{y-1}	x_{x+1},y_{y-1}
x_{x-1},y_y	x_x,y_y	x_{x+1},y_y
x_{x-1},y_{y+1}	x_x,y_{y+1}	x_{x+1},y_{y+1}

Figure 2a.
Moore Neighborhood with $r = 1$.

	x_x,y_{y-1}	
x_{x-1},y_y	x_x,y_y	x_{x+1},y_y
	x_x,y_{y+1}	

Figure 2b.
von Neumann Neighborhood with $r = 1$.

Odd x_{x-1},y_{y-1}	Odd x_x,y_{y-1}	
Odd x_{x-1},y_y	x_x,y_y	Even x_{x+1},y_y
	Even x_x,y_{y+1}	Even x_{x+1},y_{y+1}

Figure 2c.
Alternating Margolus Neighborhood

The third common neighborhood is quite a bit different. Designed to model physical systems (Lattice Gases [6]), the Margolus neighborhood has the smallest neighborhood with just 4 members. However, Margolus neighborhood automata operate with an alternating partitioning scheme [7]. The 2x2 partitioning scheme effectively groups four cells together into what can be looked at as a macro-cell. Figure 2c. shows the neighbors of the central cell with the alternating partitions indicated by the odd and even labels. Without the alternating partitioning scheme, transitions between configurations would not propagate beyond the individual partitions [7]. In a Margolus neighborhood based CA, the states of the four cells in the macro-cell indicate the current configuration of that macro-cell. For Margolus automata, the current configuration of the

macro-cell is seen as the state, and the transition function works on macro-cells as opposed to individual cells. The number of possible configurations for these two dimensional neighborhoods are $S^{(2r+1)^2}$, $S^{2r(r+1)+1}$, and 2^4 for Moore, von Neumann, and Margolus neighborhoods respectively.

For all neighbor-hooding schemes, the wrapping of border cells must be taken into consideration when determining the indices of the neighbors of the central cell [2]. Most software implementations of CA elect to emulate an infinite grid of cells by wrapping the borders. Formally, calculating the index for a neighbor could be described as a modulo operation. For example, the neighbor of the $N_{x\pm1}$'th cell would be written as $N_{(x\pm1)\%N_x}$ where % represents the arithmetic modulo operation, requiring that mod(-1,32)=31 for example. Figure 3 illustrate this idea for the Moore neighborhood with range $r = 1$.

$x_{(x-1)\%N_x}, y_{(y-1)\%N_y}$	$x_x, y_{(y-1)\%N_y}$	$x_{(x+1)\%N_x}, y_{(y-1)\%N_y}$
$x_{(x-1)\%N_x}, y_y$	x_x, y_y	$x_{(x+1)\%N_x}, y_y$
$x_{(x-1)\%N_x}, y_{(y+1)\%N_y}$	$x_x, y_{(y+1)\%N_y}$	$x_{(x+1)\%N_x}, y_{(y+1)\%N_y}$

Figure 3.
Moore Neighborhood indexing with wrapping of borders.

Though most software implementations of 2-dimensional Cellular Automata wrap the borders of the grid, special precautions must be taken for hardware implementations. In hardware, each CA cell would be an identical circuit. The cells would each have connections to their neighbors. Connecting each border cell to the other side of the grid would increase the cost and scalability of the implementation. As such, it is often

advised that CA systems destined for hardware take this into account [2]. Because the operation of the CA still requires that each cell have the same neighborhood, some special-case null-value must be chosen as transition functions for those non-existent neighbors to avoid leaking information in or out of the CA grid.

1.1.4. CA Transition Function.

The last member of the 4-tuple, the local transition function f, defines the behavior of the automata. f is directly dependent upon the number of states and the chosen neighbor-hooding scheme. Let $S^n_{x,y}$ denote the current state of the cell in a 2-dimensional grid field [2]. f transforms a cell to its next state based on it neighborhood:

$$f : S^n_{x,y} \rightarrow S^{n+1}_{x,y}$$

Let $Grid_c$ be the current configuration of the entire grid. After applying the local transition function f to all cells in the field, the global transition function F evolves [2], changing $Grid_c$ into $GridF_n$.

$$F : GridF_c \rightarrow GridF_n$$

When choosing f for applications that are information preserving, it must be specially crafted to avoid destroying information. For example, in Conway's Game of Life, which has a Moore neighborhood with range $r = 1$, a living cell with zero neighbors will die, or transition from state 1 to state 0. Since there are no complimentary rules for the spontaneous birth of a cell, Conway's rules are not information preserving. For some cryptographic purposes (i.e. not random pattern generation), f must be designed to be injective. The injective property simply requires that all inputs to the function have

7

specific outputs.   If the CA state values are based on its current neighborhood configuration, as opposed to an aggregate sum of neighbors, then creating one-to-one functions that preserve all information is simplified.   Injective transition functions inherently imply the existence of the inverse of f, f^{-1}, which defines $\mathbf{F^{-1}}$ as the inverse operation on the entire grid field. $\mathbf{F}$ is reversible if and only if $\mathbf{F}$ is a bijection. [2] If a CA requires reversible rules, then the function f must be its own inverse. For example, if configuration (or state) 'A' is changed to 'B', then 'B' must also change to 'A'. A CA with a bijective function f can be described as a reversible cellular automata or RCA.

1.1.5. Example of a Cellular Automata.

The standard example of cellular automata is John Conway's Game of Life. This automata uses a Moore neighborhood with r = 1, giving eight total neighbors to each cell for a total of nine in a neighborhood. The transition function is based on the number of 'living' neighbors and the state of the current cell. Under Conway's rules, if a dead cell has exactly three neighbors, it will be born; if a live cell has two or three, neighbors it survives to the next generation; in all other cases, the cell dies. Formally, Conway's Game of Life can be described as a 4-tuple CA = (d, S, N, f).

$\mathbf{d} = (Nx, Ny, Nz) = (X, Y, 0)$ where X and Y are natural numbers.
$\mathbf{S} = \{0,1\}$ with 0 and 1 representing dead and living cells respectively.
$\mathbf{N} = \mathbf{N}_{Moore,1}$ a Moore neighborhood with range 1.
$\mathbf{f}$ = is described by the following table.

State	Live Neighbors	Next State
0	0,1, or 2	0
0	3	1
0	4,5,6,7 or 8	0
1	0 or 1	0
1	2 or 3	1
1	4,5,6,7 or 8	0

Table 1. Conways' Transition Rules

Figure 4 illustrates five generations of Conway's rules with grey tiles representing the living cells (cells in state 1) and the numbers indicating the count of living neighbors for the chosen cell. During each generation, every cell counts the number of living neighbors in its neighborhood. Next, the local transition function is applied to each cell, which determines the next state of the cell based on its current state and its' living neighbor count. From the first randomly selected generation shown in Figure 4, all the non-living white cells with a value of three (indicating the number of living neighbors) will be born in the next generation. In the second generation, all the cells with the values four and five will die in the next generation, in addition to the usual births. The map continues to evolve via Conway's rules until it reaches a stable configuration in the fifth generation.
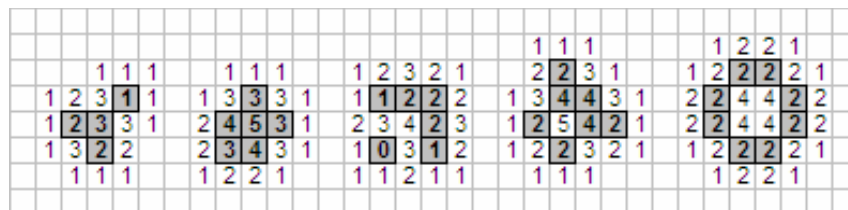
Gen: 1   2   3   4   5

Figure 4. 5 Generations under Conway's rules. Numbers show the living neighbor counts of the cell, grey cells are living.

9

1.2. Cryptography

Cryptography is the process of encoding and scrambling data for confidentiality purposes. The strength of a cryptographic algorithm (cipher) is not specifically in the algorithm itself; instead, its strength is determined by the 'secret keys' and the type of network used (SPN, feistel, data-dependent rotations) [8]. The most basic way of differentiating cryptographic algorithms is by the number of keys used. A symmetric key cipher requires only a single key to encrypt the user's data. This single key is mixed into the data in such a way that the same key is used to undo the mixing. This property implies that the way in which the key is mixed is its own inverse, and most likely the XOR operation described later. The second basic type of cipher uses two keys to do its work. These are typically called public-key cryptosystems. The operation of such two-key asymmetric ciphers are fundamentally different from their single-key cousins. Besides providing confidentiality of data, these asymmetric ciphers also offer integrity checking and verification of author for non-repudiation. Focusing on symmetric ciphers, like Rijndael, there are some basic components and methods used that require some introduction.
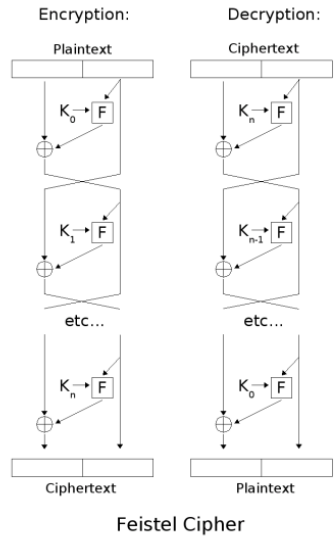
Figure 5a. A Feistel Network [10]

1.2.1. Types of Ciphers.

There are a few main structures to crypto algorithms in use today. They all seek to create what Claude Shannon calls "confusion and diffusion" of the input data [9]. Diffusion is the process of scrambling and swapping bits among bytes within a block of data. This is typically done before a confusion step. On the other hand, Confusion is performed by byte substitutions, meaning each byte is replaced by a different byte. An advantage of feistel networks is that their design is nearly identical for encryption and decryption, only requiring the key-schedule to be reversed. Figure 5a illustrates a simplified process of encrypting and decrypting a plaintext block [10]. For encryption, the block is split into two halves. One of the halves is processed by the function F. This function F performs the diffusion and confusion of the data, using whichever method the feistel cipher has chosen. Next, either as another step or as part of the function F, a portion of the key K is mixed into the data with an XOR. The resulting half block is XORed with the other half, and the process continues this way. The two half blocks

11

swap positions, and continue utilizing F and K until the entire key has been used. For

decryption, the exact same algorithm is used, but the key stream is reversed.

XOR Key and State (Confusion)

Substitute Bytes of State (Confusion)

Shift Rows of State (Diffusion)
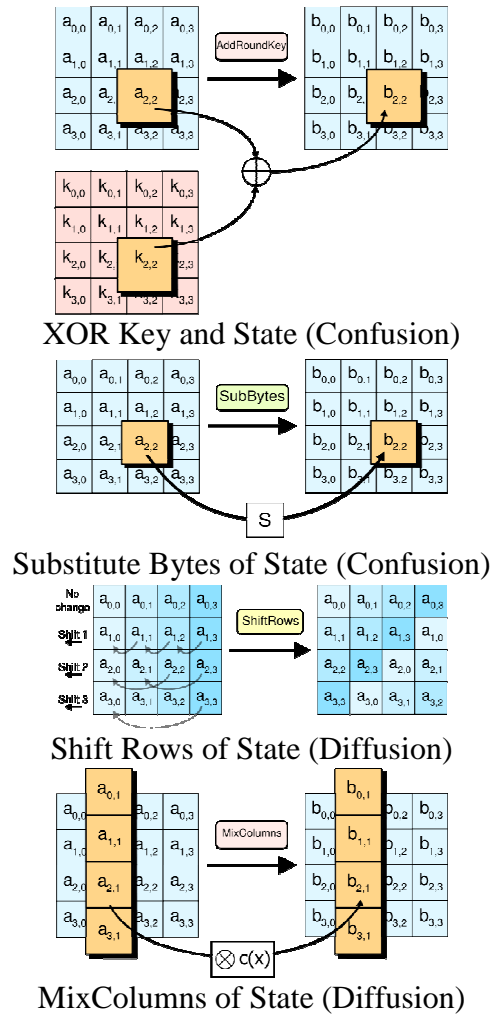
MixColumns of State (Diffusion)
Figure 5b. The four steps of AES, a SPN Cipher [12]

Substitution Permutation Network (SPN) ciphers, like Rijndael, borrow a lot from

the mixing functions of Feistel ciphers. However, the encryption and decryption

processes differ by more than a simple key reversal. Rijndael consists of steps that mix

the key, substitute bytes using an S-box, shift rows within the block and then mix

columns within each block (Figure 5b) [11]. Figure 5b, shows each of the confusion and

diffusion steps used in the SPN cipher, in no particular order. The first confusion step in

the figure shows how the plaintext block, or *state,* is XORed with some of the Key. This is represented as MatA ⊕ MatK = MatB. The second step shown in Figure 5b shows the second confusion step, called SubBytes, which is used in Rijndael and similar SPN ciphers. This process uses an S-Box to produce unique outputs for each input. The next step in the figure is simply the ShiftRow diffusion step, which deterministically rotates rows in the state, which move bits from their original location in the *state*. The second diffusion step illustrated in Figure 5b, is the MixColumns function. This function uses matrix multiplication in the Galois field to diffuse bits within each column among all four column entries. Thus, for an SPN cipher like Rijndael, confusion and diffusion functions (and their inverses) work together and in a certain order to provide security. The SPN decryption process differs from the encryption in that only the inverse functions are called and in the opposite order, which makes designing SPN ciphers slightly more involved than a Feistel cipher.

The S-box, mentioned above, is the heart of most SPN ciphers and responsible for a large amount of the confusion aspect of the cipher. S-Box is a term that simply means substitution box. An 8-bit S-box contains 256 unique entries: one for each of the 256 8-bit numbers. The input byte is used as an index into the table, and the value at that index is the substitution value. To undo the substitutions indicated by the S-box, there must also be an inverse S-box. These boxes must stay in sync (remain inverts) for encryption and decryption to work.

A	B	A ⊕ B
T	T	F
T	F	T
F	T	T
F	F	F

Table 2. Truth table for XOR operation.

A	10010011	C	00100111	C	00100111
B	10110100	B	10110100	A	10010011
A $\oplus$ B = C	00100111	C $\oplus$ B = A	10010011	C $\oplus$ A = B	10110100

Figure 6. Example of XORs properties with an 8-bit string.

<u>1.2.2. Basic Boolean Operations.</u>

Both SPN and Feistel networks (and nearly all ciphers in existence) depend heavily on the XOR operation to do most of their invertible operations. XOR ($\oplus$), which means Exclusive OR, is a Boolean operation that results in true if and only if one of the inputs is true (Table 2). This exclusive disjunction allows two pieces of data to be mixed invertibly as long as one of the original pieces of information is known. Thus, if A $\oplus$ B = C, then C $\oplus$ B = A, and C $\oplus$ A = B (Figure 6). At the simplest level, all encryption algorithms XOR the plaintext and the key together to get the cipher text. This naïve example has the obvious problem that if an attacker were able to guess the content of a plaintext block, they could retrieve the encryption key. Thus, algorithms have diffusion aspects as well. These properties and the performance of bit-level operations make XOR a perfect fit with cryptographic applications.

Electronic Codebook (ECB) mode encryption

Electronic Codebook (ECB) mode decryption
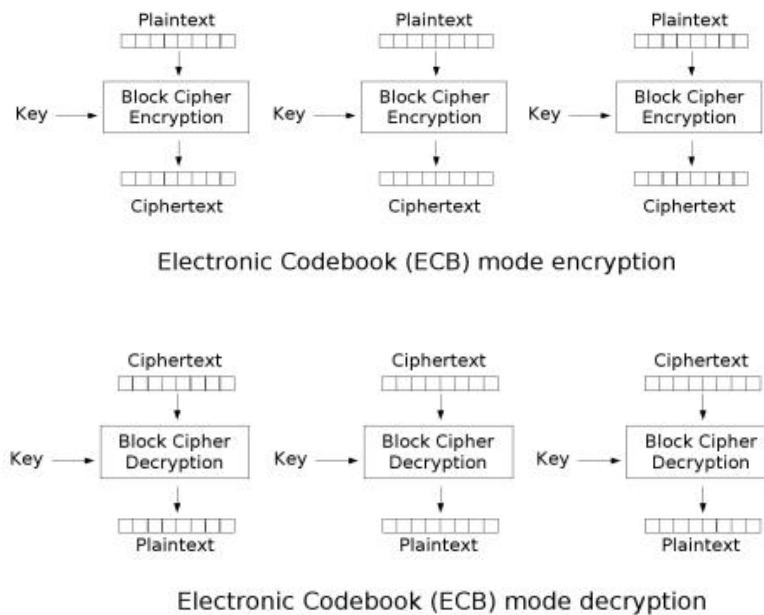
Figure 7. ECB block mode operation [12]

1.2.3. Block Cipher Modes of Operation.

All block ciphers operate on fixed sized chunks of data, or blocks. Because of this, any two identical blocks encrypted under the same key will output the same cipher text. Since this is a flaw inherent in any block cipher, a number of "modes" exist to increase the security implementations of block ciphers. In Electronic Codebook (ECB) mode (Figure 7) [12], each input block is encrypted in turn. A block of data is first read into a buffer. The encryption algorithm is then performed on that buffer, which mixes the key into the data. This processed block is then output to the cipher text file, and the next plaintext block is encrypted in turn. For example, if you were to encrypt a large image with ECB mode, the input blocks for identical regions would show through in the cipher text (Figure 8) as per the inherent flaw. ECB is not recommended for any cryptographic implementation because of this reason [13].

15

Scaled Original   Scaled RC5 Encrypted ECB   Scaled AES CBC
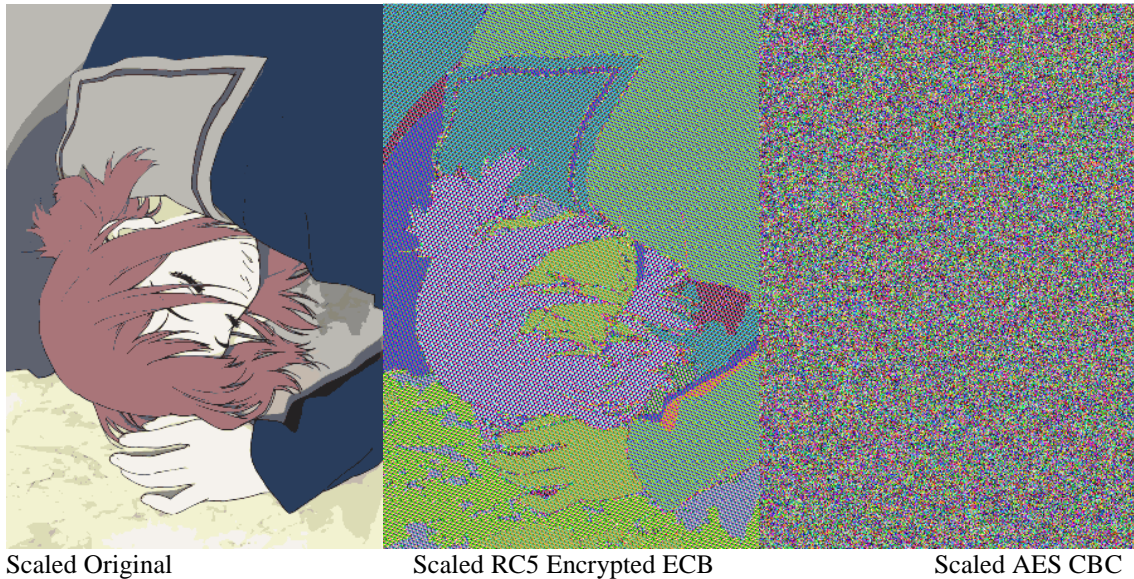
Figure 8.

Comparison of ECB and CBC mode for a 6885 x 10000 pixel image. (Non-Interpolated Post Scaling)

       A much better choice of mode for a block cipher is the Cipher Block Chaining (CBC) mode. Figure 9 illustrates this mode. CBC mode requires an initialization vector (IV), which is simply a random block of data that will be XORed with the plaintext before encrypting a block. To create an IV, first fill a block-sized buffer with random data from any random number source. The second step is to encrypt this IV with the users' key. The resulting IV should be uniform random after encryption. The IV does not need to be kept secret; it only needs to be random and never re-used for the same key. The initial IV is now ready for use in processing plaintext blocks. It is used and updated exactly as shown in Figure 9 [12]. First, a plaintext block is read in from the file, the plaintext block is modified by XORing with the IV. The block is then encrypted, and the IV is set to the values of the encrypted block. This process continues until the file is exhausted. However, in order to decrypt a file with CBC mode the initial IV needs to be stored as the first block in the cipher text.

The main advantage of CBC over ECB is that all previous blocks affect the current block in one way or another [13]. Thus, if a single bit is changed in the first block of the file (and the algorithm has good per block avalanche effect), every other block will also be changed. There are other modes for block ciphers like Cipher Feedback, Output Feedback, and Counter mode, but most of them focus on creating stream ciphers from block ciphers, which is outside of the scope of this thesis.

Figure 9. CBC block mode operation [12]

1.2.4. Cryptographic Keys and Keyspace.

The *key* itself is one of the most important parts of any algorithm, as it provides 'instructions' for encrypting and decrypting the data [8]. In any symmetric cryptosystem, the key must be kept secret by the user. Typically, keys range from 64-256 bits, with more bits suggesting more security. The *key space* is the number of possible keys that could be used for input. For a key of 256 bits, there are 2^{256} available keys. The most obvious step toward security is removing the possibility for someone to do a brute force

attack in which the attacker tries many keys until a plain text matching a known distribution is generated. Statistically, a brute force attack would need to try half of the key space before finding a match, or $2^{(256-1)}$ trials for a 256-bit key. Due to the enormous number of keys to try (Table 3), it is usually computationally infeasible to use brute force on most of today's algorithms.

KeyBits	Total Keys	Single (1)	Distributed (1mil)
1	2	1.15741E-11	1.15741E-17
2	4	2.31481E-11	2.31481E-17
4	16	9.25926E-11	9.25926E-17
8	256	1.48148E-09	1.48148E-15
16	65536	3.79259E-07	3.79259E-13
32	4294967296	0.024855135	2.48551E-08
64	1.84467E+19	106751991.2	106.7519912
128	3.40282E+38	1.96923E+27	1.96923E+21
256	1.15792E+77	6.70093E+65	6.70093E+59
512	1.3408E+154	7.7591E+142	7.7591E+136

Table 3. Relationship between keylength, total keys, and days to test 50% of keys at 1 per μs and at 1 per picosecond.

Internally, keys are usually larger than the user supplied key due to key-expansion or key-scheduling functions built within a strong cipher [14] [15]. Such key expansion functions also serve another important role, by avoiding weak keys. In some algorithms, there exist key patterns that could output part of the message or part of the key in the cipher text. Or even cause no encryption to be done at all. These key expansion algorithms typically do not directly increase key-space, as they are deterministic and based on the users small input key. However, key expansion allows an algorithm to perform more *rounds* of encryption on the data with a different key each time. In Rijndael/AES, key expansion relies on the S-box and is therefore related to the S-boxes current configuration, yet it is still deterministic.

One place that researchers have attempted to integrate CA with Cryptography, is in key-expansion. The authors of [1] [16] [17] [18] all used one-dimensional CA to

generate a key-schedule based on the pseudo-random pattern properties of many one-dimensional CA rules, as illustrated in Figure 10. Thus, the first line of pixels in Figure 10 represents the first generation, or the user supplied key. Subsequent generations provide keys further in the key-schedule, which are claimed to be pseudo-random. While a truly random expanded key would b ideal for encryption (like a one-time-pad cipher), pseudo-random is the norm as most algorithms use the user-supplied key as an initialization vector for key expansion.
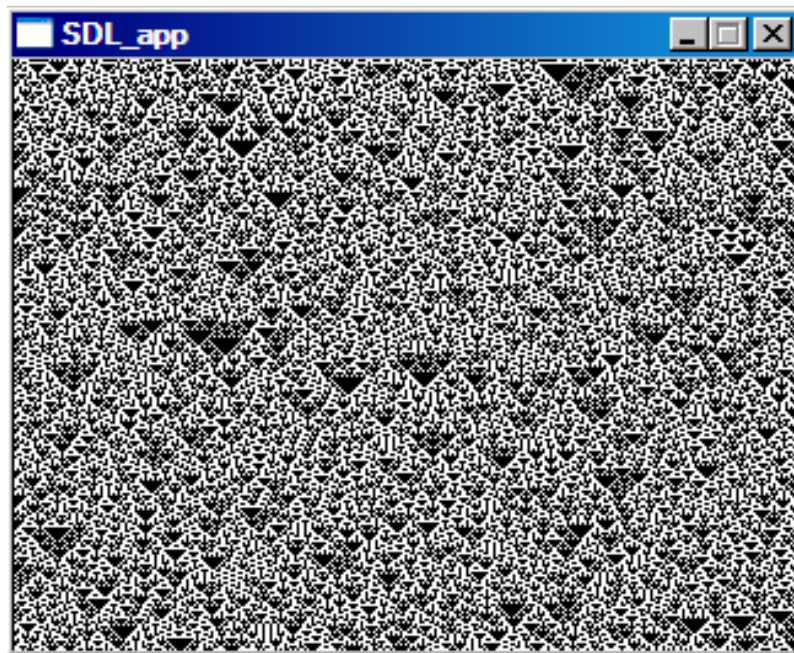
Figure 10. Pseudo-random pattern generation from a 1-D Automata Rule

1.2.5. Other Parameters of Encryption Algorithms.

Many symmetric key ciphers have parameters for the number of rounds, block size, word size, and for CA based systems the number of generations. A user modifiable block size, for example, is often a parameter of a good algorithm and allows modifications to be made based on memory and time constraints. Typically, a larger block size will encrypt faster, but block sizes directly depend on the word size the

algorithm or user supplies and their architecture. For example, the RC5 algorithm parameterizes word size to allow 16, 32 or 64 bits of data for each entry in the block. Providing parameters to the internal workings of the algorithms effectively increase the key-space, as the attacker must know these values exactly in order to decrypt a file. Another common parameter, the number of rounds, determines the number of times the encryption functions are performed on a block of data. In RC5, this parameter is variable from 1-255; in Rijndael, it is typically a function of the key-length but can be parameterized. Increasing the number of rounds increases the convolution of the data.

1.2.6. Padding Methods for Block Ciphers.

Any block cipher using ECB or CBC modes must pad the plaintext to fill out the block before encryption; the other modes require no padding, as they are stream ciphers. If the final block is not padded prior to encryption, then it will not decrypt properly. One typical method of padding suggests filling the first unused byte with 128 (0x80, or 10000000 binary) and all other bits/bytes with zeros. The other primary method is to fill all N_p bytes with the value N_p; thus if you have four padding bytes they will all be padded with 04, as shown in Figure 11, Case 1, Method 1. However, there is a problem with this scheme. If the plaintext ends on a block boundary and has the last bytes (0x01) or (0x02, 0x02), or even (0x03,0x03,0x03) then these bytes will be indistinguishable from padding and will be removed as padding. This is illustrated in Figure 11, Case 2, Method 1. Thus, it is highly recommended to add a full block of padding to the cipher text in addition to filling out the last block with padding bytes. In this scheme, all padding bytes are filled with the value, N_p + BlockSize. This allows the padding to be properly

removed after decryption and is illustrated in Figure 11 in Method 2 for both cases. This method will always work unlike Method 1.

Case 1
Padding Needed

	0	1	2	3	4	5	6	7
0	P	A	D		T	H	I	S
1		M	E	S	S	A	G	E
2		T	O		3	2	-	B
3	Y	T	E	S				

Message that needs 4 bytes padding.

	0	1	2	3	4	5	6	7
0	50	41	44	32	54	48	49	53
1	32	4D	45	53	53	41	47	45
2	32	54	4F	32	63	62	2D	42
3	59	54	45	53	04	04	04	04

Method 1. Padding with Np = 04h Depadding will remove 4 bytes.

	0	1	2	3	4	5	6	7
0	50	41	44	32	54	48	49	53
1	32	4D	45	53	53	41	47	45
2	32	54	4F	32	63	62	2D	42
3	59	54	45	53	24	24	24	24
1	24	24	24	24	24	24	24	24
2	24	24	24	24	24	24	24	24
3	24	24	24	24	24	24	24	24
4	24	24	24	24	24	24	24	24

Method 2. Padding with Np = 24h Depadding will remove 36 bytes.

Case 2
No Padding Needed

	0	1	2	3	4	5	6	7
0	50	41	44	32	54	48	49	53
1	32	4D	45	53	53	41	47	45
2	32	54	4F	32	63	62	2D	42
3	59	54	45	53	21	03	03	03

Message that needs 0 bytes padding.

	0	1	2	3	4	5	6	7
0	50	41	44	32	54	48	49	53
1	32	4D	45	53	53	41	47	45
2	32	54	4F	32	63	62	2D	42
3	59	54	45	53	21	03	03	03

Method 1. Padding with Np = 0h Depadding will remove 3 bytes.

	0	1	2	3	4	5	6	7
0	50	41	44	32	54	48	49	53
1	32	4D	45	53	53	41	47	45
2	32	54	4F	32	63	62	2D	42
3	59	54	45	53	21	03	03	03
1	20	20	20	20	20	20	20	20
2	20	20	20	20	20	20	20	20
3	20	20	20	20	20	20	20	20
4	20	20	20	20	20	20	20	20

Method 2. Padding with Np = 20h Depadding will remove 32 bytes.

Figure 11. Two padding methods.

1.2.7. Common Cryptanalysis Techniques.

Besides ensuring an adequate key space, there are many experimental methods for analyzing the general strength of a cipher. These methods can target weaknesses in the algorithmic design, software implementation, or hardware implementations. The most basic test of the algorithm is a data histogram. After processing a block or file of data, there should have been adequate confusion and diffusion to produce a uniform frequency

histogram for all byte values, regardless of the plaintext input. This uniform property is necessary to avoid first order language attacks which are based on the probability and statistics of the original data, like frequency of letters in the English language. Obtaining this uniformity even with the use of a uniform input key and uniform plaintext is the first step to showing a cipher may not be trivially breakable.

Another property of all modern algorithms is the critical avalanche effect (CAE). CAE says that for a single bit change in input byte, at least 50% of bits should change in the output byte [2]. In the AES/Rijndael algorithm, this is handled by the diffusion steps, namely the ShiftRow and MixColumn operations. The MixColumn operation specifically performs matrix multiplication using a matrix [19] which ensures that every byte of the input affects all four bytes of the output. This method is elegant, and its affine property does not weaken the cipher. These types of operations, coupled with an S-box, provide the critical avalanche criterion and byte frequency uniformity. An algorithm design that meets these criteria – in an intelligent order – has the possibility of being a secure algorithm.

Many attacks focus on weaknesses in the implementations of algorithms as opposed to the actual design of the algorithm. For example, in AES and Rijndael S-boxes are supposedly generated in such a way so as to have good non-linearity properties. However, According to [20] and [21], the S-boxes generated by Rijndael's algorithms only have 9 algebraic terms, which may prove to be vulnerability in the design. It is also suggested in [22] that a static S-box – which most implementations store in the binary – allows an attacker to simply modify the binary and discover the secret key and plaintext.

The authors argue that most implementations of ciphers with static S-boxes are vulnerable to blanking, resulting in key discovery. In addition to their examples and proofs of AES's vulnerability to this scheme, they also suggest that all SPN ciphers with "Unprotected Implementations, Static S-boxes in cipher round, Key whitening (XOR) operation after the final round, A suitable round key expansion," are likely vulnerable to S-box Blanking. In the case of AES, if an attacker were to blank the S-box with zeroes on the victims machine, then due to the XORing of the data with an entry from the key schedule, the key would be directly outputted in the cipher text. Kerins and Kursawe offer methods to defend against S-box blanking, such as physical protection of the cipher binary by the operating system, code obfuscation or executable packers, dynamic static S-box generation at run time, and dynamically modifying a generated S-box. These suggestions are useful, as the dynamic modified generated S-box offers protections against both S-box blanking as well as the weak 9-term arithmetic complexity of the standard AES S-box as mentioned in [20] [21].

<div align="center">1.3. Cellular Automata Properties for Cryptography</div>

There are many properties of Cellular Automata that may help or hinder their usefulness in cryptographic applications. For example, any cellular automata that is not injective is said to be a non-reversible CA [2]. Such a non-reversible CA loses data each generation, which lowers its usefulness for encryption purposes. It may be possible to find a use for these automata where reversibility is not important, such as random number generation and hashing functions. Yet in the case of reversible cellular automata (RCA), it is easy to see one simple use in cryptography. An RCA could be used as a diffusion step. Each cell would represent a bit from a file, and the global transition functions **F** and

<div align="center">23</div>

F^{-1} would be used for performing diffusion of bits within a block of data. However, running a RCA on a block of data for an arbitrary number of generations does not make a cryptographically secure algorithm. The use of an RCA as the 'diffusion' step in a substitution permutation cipher (SPN) seems obvious, but a single static RCA rule does not offer much more diffusion than the typical RowShift and ColumnMix operations. Even with the existence of techniques for creating RCA (tiling techniques, and second order CA) [23] the information preserving property of RCA may be of more interest for some cryptographic applications than reversibility. A final property – and one of the most critical for cryptographic purposes – is the affine property. If a Cellular Automata Cryptosystem (CAC) depends heavily on weak affine transformations, the low linear complexity of the transformations will most likely lead to an insecure algorithm when used as a diffusion step [24].

Figure 12. Defining rules for 1-Dimensional automata.

1.3.1. Previous Research in Combining CA and Cryptography.

As mentioned previously, there have been many attempts at creating a CAC. In [16] S. Nandi, et al, propose a class of block and stream ciphers based on what they call "fundamental transformations". The scheme offered creates an alternating group of permutations using EXNOR logic, various programmable one dimensional cellular automata (PCA) rules (51, 153, and 195), and a rule selection function based on a given key. One-dimensional automata rules were defined by Stephen Wolfram [25], as shown

in Figure 12. The three top cells indicate the current configuration of the central cell and its two neighbors. The single cell in the second row indicates what color the central cell will have in the next generation. The binary pattern of the next generation defines the rule number from 0 to 255. For example, in Figure 12, the top example has a next generation pattern of 00000000, or rule 0, while the bottom example has the pattern 00110011, or rule 51. A rebuttal to their proposal is offered in [26], where Blackburn and Murphy rigorously analyze the scheme and conclude that all the transformations are of the affine group and are therefore cryptographically weak. They state that they can determine the initial state of the automata (in this case, the key) by solving a set of linear equations with $L - 2$ variables and with only 2^{L+2} trials, where the initial key is an LxL bit matrix and the proposed value of L by Nandi is 16.

Nandi, et al, also claim the ability to use PCA as "high quality pseudorandom pattern generators" as well as for dynamic key generation and manipulation for stream ciphers. Their proposed use of CA as pseudo random pattern generators was not questioned and is supported by the finding in S. Wolfram's research [18]; yet [24] argues that the Komogorov complexity theory proves that the simple local rules of cellular automata cannot create true randomness and thus may not be good enough for cryptographic applications. Nandi, et al, question the absolute terms of insecurity based on their affine transformations, inquire about ways to absolutely secure such a system, and re-iterate their correct statements about CA natural fit with VLSI design in [27].

In his paper "Cryptosystems Based on Reversible Cellular Automata", Jarrko Kari offers another CA based cryptosystem which offers some useful suggestions to any future attempts. He proposes both a secret-key and a public-key cryptosystem using cellular

automata. The proposed symmetric key system uses a CA to modify the key each block. Kari suggests that each application of the CA to the key offers increased security. As with most systems, special precautions must be taken to avoid weak keys. A simple solution to the problem of homogenous input key or plaintext is offered. Kari suggests always forcing some of the cells to certain states and using input data to populate the other cells. This suggestion and parameterization of the k value offer an avalanche effect that results in largely different output if a single bit of the key or plain text is changed.

Kari also offers an asymmetric or public key CAC. The general theory of public-key encryption using CA is to craft a complimentary pair of automata that are hard to find the inverses of individually. Finding the inverse of a given automata, is an NP-Hard problem according to Toffoli and Margolus in [23]. However, going against traditional methodology, Kari suggests the use of many simple self-inverting "marker automata", or transformations, which will decrypt when run in reverse. This sounds more like a single key system, as the key is simply the order in which to run the data through the "automata" (forward or backward). Calling these simple and linear transformation rules an automata, let alone a crypto system, is misleading.

In [1], the authors propose a private key block cipher system that moves away from simply using a 1-dimensional CA to generate the key stream and XOR logic as previous research in [17] and [16] did. The algorithm's key consists of the number of RCA, the iterations p, and p input vectors. The two transition matrices generated based on this data are inverses of each other and have possible uses in encryption. However, the transition matrices generated are assumed (by the author) to be linear, which may prove to be a weakness. The authors suggest that their algorithm is more resistant to

exhaustive search attack than other CACs due to its larger key space lower bound of 2^{2p}. As most other offered solutions, their real focus is on fast parallel operation for hardware VLSI designs. As such, their boundary neighbors are always considered with a null state as previously suggested.

Efforts to use CA for cryptography seek to take advantage of the fact that simple rules can generate complex pseudorandom patterns and that they can be implemented in hardware for very fast and efficient operation. While there have been many attempts in using CA for cryptography, not many have been proven to be cryptographically secure. On the contrary, any cryptosystem based on affine, linear, simple, or "fundamental transformations" is cryptographically weak and only a problem of solving a system of equations with the support of trial data. Using one-dimensional pseudorandom patterns generated by CA for a key stream is argued to be insufficient for cryptography [24]. The use of non-homogenous cellular automata for public-key systems also suffers from the challenge of generating complimentary pairs of automata and the fact [24] that NP-hardness does not necessarily guarantee security for cryptosystems.

1.3.2. Other Possible Uses of CA in Cryptography.

Researchers have made many attempts at creating cryptosystems based nearly entirely on cellular automata. Some of the problems in these CACs lead researchers to question the true applicability of CA for any cryptographic use. While the idea of using CA as a cryptographic primitive, like a key expansion algorithm, or as the confusion and diffusion steps themselves may seem like a lost cause, CA may still have some possible application. For example, in [28] the author proposes the use of CA to generate new and unique S-boxes for block ciphers. While it remains to be shown that CA can generate

good non-linearity in an S-box, S-boxes are a proven and very powerful cryptographic primitive. In this sense, while developing entirely new algorithms based on cellular automata might be questionable based on prior attempts, CA surely have some use in further enhancing existing and proven cryptographic primitives like an S-box. While S-boxes are typically generated deterministically, or even hard-coded into the binary, there is an opportunity for increased security in dynamic S-boxes that are kept secret from the attacker.

Thus, based on the conclusions of [20] [21] [22] and suggestions of [28], the S-box cryptographic primitive, and specifically the AES/Rijndael implementation, could possibly be improved in the following ways:

1. Dynamically generate S-boxes at runtime, do not store a single static S-box in the binary. Dynamic generation protects against S-box blanking.

2. The 9-term algebraic expression for the S-box could be removed with some dynamic modifications of the S-box.

3. Having more than one initial S-box based on user choice or key, as opposed to AES's single static S-box, increases complexity. S-Box choice also modifies the key-expansion process.

2. CRYPTOSYSTEM DESIGN AND PROCEDURES

Initially there were several ideas that could have been used to develop this cellular automata cryptosystem. The base encryption algorithm chosen for the cryptosystem is Rijndael with some restrictions placed on its parameters. The chosen parameters guarantee a very high level of security and allow for a simpler implementation. The choice of CA for S-box mixing was based on the simple Margolus neighborhood. Margolus, which was the first cellular automata investigated for this purpose, ended up being a very suitable method due to the simplicity and elegance of its alternating partitioning scheme. The following section provides details on the design and implementation of the proposed CAC.

This thesis proposes a CAC based on Rijndael symmetric block cipher that implements:

1. Dynamic S-box generation, generating up to 128 unique and valid initial S-boxes.

2. A Cellular Automata that dynamically, reversibly, and uniformly redistributes S-box and Inverse S-box values.

3. The option of modifying the S-box before key expansion for generating different expansions based on the input parameters.

4. User parameters controlling the Initial S-box choice and the number of generations the Automata should modify the S-box per block, as well as a threshold parameter of the CA.

5. 256-bit block size, 256-bit key size, CBC mode with IV generated from clock drift calculations, double padding.

6. Optional key shuffling CA is also provided.

7. All user input parameters must be known to properly decrypt a file.

There are many steps in developing and testing a CAC. In this system, there are two main parts: the encryption system and the S-box modifying CA. Each part has multiple steps; thus a modular unit-tested design was used throughout. Each function was rigorously tested during the implementation to ensure the proper operation and to avoid debugging nightmares where encryption and decryption just do not work properly.
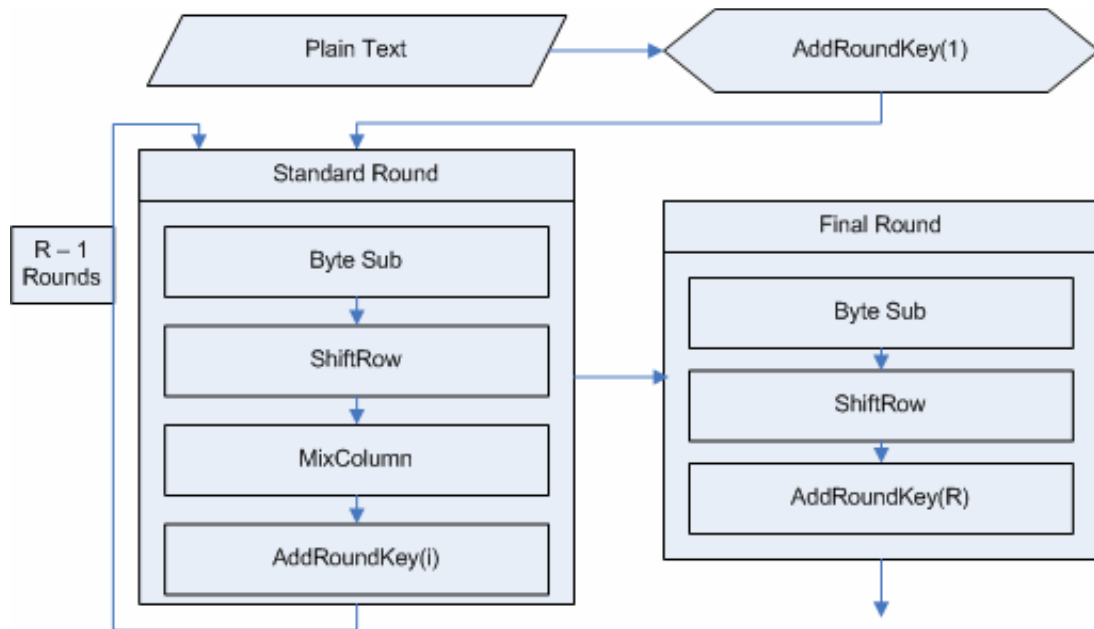
Figure 13. Rjindael Encryption Flowchart for processing a single block. doRounds

2.1. Rijndael Implementation.

Implementing Rijndael with specific parameters of 256-bit key and 256-bit blocks was the first step towards realizing the design of the eventual CAC. A systematic approach was taken, with each function being tested and verified before moving to the next (Figure 13). The Galois mathematics functions were based on and verified with the example C++ code, tutorials, and lookup tables released into the public domain on Sam Trenholme's website [19]. Specifically, the public domain Rijndael code utilized in the CAC deals with generating multiple supporting tables. This code was utilized because it is highly optimized for performing various Galois Field mathematics in $GF(2^8)$, and to re-write these base functions into less optimized and less tested versions would be counterproductive.

Specifically, the public domain functions that were implemented generate the log table, exponentiation table, division table, and the key expansion table. These initialization functions generate 8-bit 256 entry tables for performing $GF(2^8)$ math quickly. These functions were modified at various times during their integration into a working Rijndael implementation. Most of these changes were data type and data structure changes, for example making all data types unsigned 8-bit (for the 8-bit field) and making all two dimensional tables one dimensional. Another function that utilizes public domain C++ code is the mixColumn operation. Once again, the existence of high quality public domain source code for this crucial step in implementing the Rijndael half of the CAC makes re-implementation pointless. The public domain functions were mainly used for convenience in the early stages of development, and on their own they do

nothing. They serve as initialization and mathematics routines only. Charts showing functions source and their use in the process can be found in Appendix A5 and A6.

2.1.1. Generating Rijndael's Galois Field.

The first step of the Rijndael implementation design dealt with generating the Galois Field used throughout the AES/Rijndael algorithm. A Galois Field is typically described with the notation $GF(p^n)$ where p is the characteristic prime number, and the value (p^n) represents the order (total elements) of the field. In the case of AES\Rijndael, the field used has order 256 and characteristic prime of n=2, thus a $GF(2^8)$. Doing mathematics in a Galois field requires that all results fit within the field, i.e. 8-bits. Adding and subtracting in Rijndael's Galois field are represented by an XOR operation. In contrast, multiplication, division, exponentiation, and logarithms require operations that are more complex. Luckily, these operations can be handled with lookup tables. In the implementation, five tables are generated based on a user chosen Galois generator. These generators are numbers that , traverse all possible values in the Galois field (except zero) when exponentiated 255 times. There are 128 Galois generators in $GF(2^8)$. The chosen Galois generator affects all five look-up tables in the implementation, the S-box, and the later key-expansion. Making it a user parameter increases key-space and hides the initial S-box of the CAC from the attacker, as opposed to AES's single S-box.

2.1.1.1. Multiplying in $GF(2^8)$.

As mentioned, multiplication in $GF(2^8)$ is more complicated than the simple XOR that is used for addition and subtraction. The abundant Galois field theory and supporting information in the AES proposal for Rijndael [9] can be implemented with a surprisingly simple algorithm, shown in Figure 14:

```
set product = 0
set highbit = 0
for x = 0 to 7
  if(operandB AND '01')
    product = product ⊕ operandB
  highbit = (operandA AND '80')
  bitShiftLeft(operandA, 1)
  if(highbit)
    operandA = operandA ⊕ '1b'
  bitShiftRight(operandB,1)
return product
```

Fig. 14 Multiplying in Rijndael's Galois Field

2.1.1.2. Generating the Exponentiation and Log Tables.

The generator is first used to build the exponentiation table and the log table as

per the following algorithm shown in Figure 15:

```
set expTable[0] = 1
set expTable[255] = 1
for x = 1 to 255 do
  set expTable[x] = expTable[x-1] * galoisGenerator (galois multiplication)
  set logTable[expTable[x]] = x
set expTable[255] = 1
```

Figure 15 Generating the Exponetiation and Log Tables

The exponentiation table and the log table can be used to multiply two numbers in

the Galois field much more quickly than the standard Galois multiply (which was used to

generate the tables). This is done with the method shown in Figure 16:

```
set a = logTable[X]
set b = logTable[Y]
set sum = a + b mod 255  (normal addition, not galois)
set product = expTable[sum]
```

Figure 16 Multiplication of X and Y in the GF using tables.

33

Division in Rijndael's Galois Field is performed by taking the logarithm of the numerator and subtracting the logarithm of the denominator modulo 255 and looking it up in the exponentiation table. The only division operations actually performed in Rijndael always have one as the numerator; thus, we generate the multiplicative inverse table as shown in Figure 17. The log of 1 (the numerator) always has a value of 255. The log of the denominator is subtracted from 255, and the result is looked up in the exponentiation table. These tricks for quickly performing mathematics in the Galois Field were found on Sam Trenholme's website, and replaces the typical generalized Galois Field polynomial based calculations for table look ups. These tables will be used to build the initial S-box and inverse S-box.

```
set mulInv[0] = 0
for x = 1 to 255
  mulInv[x] = expTable[255 – logTable[x]]
```

Figure 17 Generating the Multiplicative Inverse

2.1.1.4. Generating the S-box and Inverse S-box.

The S-box, handles Shannon's confusion step of the SPN cipher Rijndael. It simply handles all byte substitutions and thus requires an inverse table. It is generated by taking the multiplicative inverse of a given number and transforming it with a simple affine transformation matrix. A single value for the S-box can be transformed and calculated with the algorithm shown in Figure 18:

```
set s = mulInv[inputByte]
set x = mulInv[inputByte]
for c = 0 to 3
  s = ROTL(s,1)  (circular rotate s left by one)
  x = x ⊕ s        ( ⊕ = XOR)
outputByte = x ⊕ galoisGenerator
```

Figure 18 Getting an S-box Entry (SUB)

Using the SUB algorithm (Figure 18) the S-box and Inverse S-box tables were generated, as shown in Figure 19:

```
for x = 0 to 255
  Sbox[x] = SUB(x)
  SboxInv[Sbox[x]] = x
```

Figure 19. Generating the S-box and Inverse S-box Tables

2.1.2. Performing the MixColumn Operation.

The MixColumn operation performs half of Shannon's diffusion step in Rijndael, with ShiftRow performing the other half. It accomplishes this diffusion of bits by using matrix multiplication within the Galois field.   Figure 5b, showed a graphical representation of this function. The MixColumn operation is performed on every column of the *state*. The state represents the current block of data being operated on. In this case, the 256-bit block size being implemented has a state of 4x8 bytes. Each of the eight 4x1 column matrices are multiplied by one of the two 4x4 matrices shown in Figure 20, depending on whether encryption or decryption is being performed. Multiplying by these matrices ensures that all four entries in the column will mix with each other and reversibly so. The creators of Rijndael chose these matrices based on the fact that within the Galois field the columns are considered polynomials over $GF(2^8)$, which are multiplied modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ [9]. This constant

35

polynomial is co-prime to $x^4 + 1$ and thus invertible as per standard Galois Field methodology [9], which this thesis largely ignores with its optimized table based approach to Galois Field mathematics.

```
2   3   1   1          14   9  13  11

1   2   3   1          11  14   9  13

1   1   2   3          13  11  14   9

3   1   1   2           9  13  11  14
```

Fig 20. Rjindael's 4x4 Matrix   Rjindael's Inv 4x4 Matrix

Multiplication and addition during the matrix operations are performed within the $GF(2^8)$, with addition and subtraction being the XOR operation [9]. These kinds of operations quickly lead toward meeting the Critical Avalanche Criterion [29] which requires that any bit change in the input affects an average of 50% of the bits in the output. The MixColumn operation moves a fair number of bits to different positions, as can be seen in Figure 21. Figure 22 shows the original pattern representing the four rows of data shown in Figure 21, and the bit pattern after performing the MixColumn operation.

```
00 08 10 18 20 28 30 38      80 88 90 98 a0 a8 b0 b8
40 48 50 58 60 68 70 78   ➤  db d3 cb c3 fb f3 eb e3
80 88 90 98 a0 a8 b0 b8      00 08 10 18 20 28 30 38
c0 c8 d0 d8 e0 e8 f0 f8      5b 53 4b 43 7b 73 6b 63
```

Fig. 21 MixColumn Effects in Hexadecimal on an input block.

36
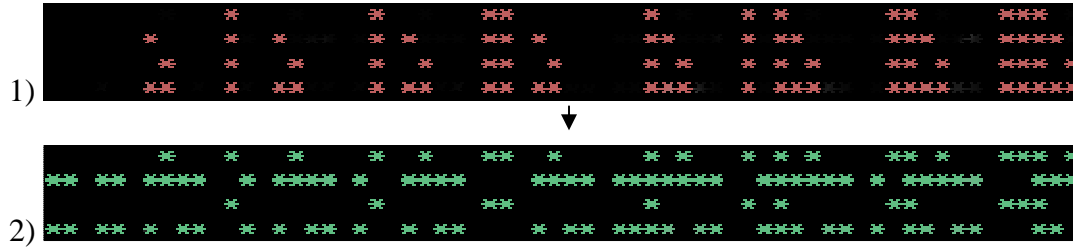
Fig 22. Mix column effect on the bits of the state shown in Figure 21.
1) Original Bit Pattern, 2) Resulting MixColumn Bit Pattern

2.1.3. Performing the ShiftRow Operation.

The ShiftRow operation performs the second half of the diffusion of bits throughout the *state*.   The prime operation of ShiftRow is a one-byte circular shift of the row: left or right for encryption or decryption respectively. The number of times each row is shifted depends on the block size and the row number itself. For the fixed 256-bit block size being implemented, the 8-byte rows 3, 2, and 1, are shifted 4, 3, and 1, times respectively. Figure 5b, showed a simple graphical example of this process. Row 0 is never shifted with any block size. These operations result in yet more diffusion, as shown in Figure 23 and Figure 24. You will notice that the first row remains unchanged, the second row was shifted once, and the third and fourth rows are shifted 3 and 4 times respectively. Figure 24 shows the data from Figure 23 represented as bits, with an asterisk representing a bit that is turned on.

```
00 08 10 18 20 28 30 38      00 08 10 18 20 28 30 38
40 48 50 58 60 68 70 78      48 50 58 60 68 70 78 40
80 88 90 98 a0 a8 b0 b8  ➜   90 98 a0 a8 b0 b8 80 88
c0 c8 d0 d8 e0 e8 f0 f8      d8 e0 e8 f0 f8 c0 c8 d0
```
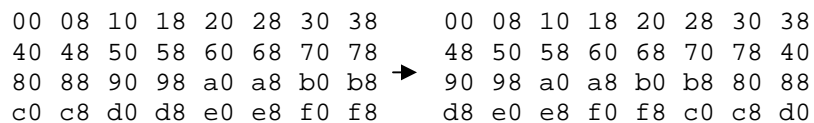
Fig. 23 ShiftRow Effects in Hexadecimal on an input block

37
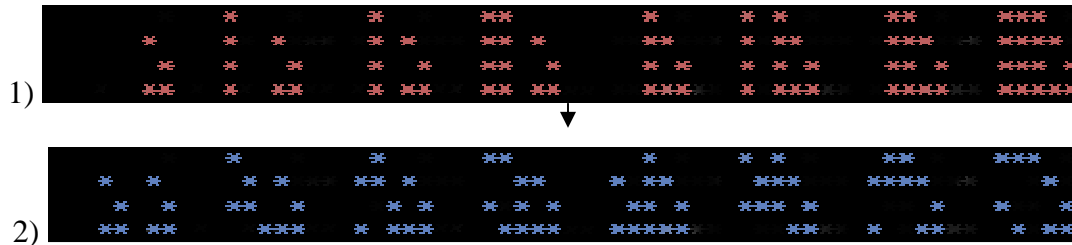
Fig 24. ShiftRow effects on the state shown in Figure 23.
1) Original Bit Pattern, 2) Resulting ShiftRow Bit Pattern

2.1.4. Performing Key Expansion.

Key expansion is the process of taking the user's small and possibly weak input key and procedurally expanding it into a much larger and stronger key. The key expansion functions for Rijndael differ slightly based on the size of the input key. For a 256-bit key, the algorithm generates an additional 448 bytes for the key schedule. In this implementation, forcing a fixed sized 256-bit key slightly simplifies the process of generating the key schedule. However, the process still requires several functions. These functions consist of an 8-bit circular rotate on a 32-bit word, the RCON operation (exponentiation of 2), a substitution using the S-box, and finally the key schedule function. While the first two functions may be trivial (Figure 25, 26), they are illustrated in pseudo-code along with the others for completeness.

```
set a = in[0]
for c = 0 to 3
   in[c] = in[c+1]
in[3] = a
```

Fig. 25 Circular left rotate of a four byte word.

```
set result = 1
if(inputExponent == 0) return 0
while(inputExponent != 1)
   result = gmul(result, 2)
   inputExponent = inputExponent – 1
return result
```

Fig. 26 Rcon operation, exponentiation of 2 in GF(2^8).

The ScheduleCore function (Figure 27) does most of the work during key expansion. It takes four input bytes (32-bits) of the key and uses the rotate and RCON operations to do some mixing of the key. This mixing is necessary to avoid weak keys and weak expansions.

```
rotate(FourInputBytes)
for c = 0 to 3
   FourInputBytes[c] = Sbox[FourInputBytes[c]]
FourInputBytes[0] = FourInputBytes[0] ⊕ rcon(inputExponent)
```

Fig. 27 ScheduleCore scrambles four bytes and uses rcon on the first byte.

The final algorithm (Figure 28) in key expansion uses ScheduleCore to do most of its work. However, in the case of 256-bit keys, this algorithm also does some extra mixing by adding an extra S-box substitution. This extra step is necessary to maintain strength in larger input keys that is not required for small keys. For the 256-bit key expansion, the ScheduleCore algorithm expects the first 32-bytes of the 480-byte InputKey array to contain the user supplied key. This expectation allows in place expansion to the final 480-byte key.

```
array FourTempBytes[4]
set keyOffset = 32
set inputExponent = 1
while(keyOffset < 480)
  for a = 0 to 3
    FourTempBytes[a] = inputKey[a + c – 4]
  if( c mod 32 == 0)
    ScheduleCore(FourTempBytes, inputExponent)
    inputExponent = inputExponent + 1
  if( c mod 32 == 16)
    for a = 0 to 3
      FourTempBytes[a] = Sbox[FourTempBytes[a]]
  for a = 0 to 3
    inputKey[c] = inputKey[c – 32] ⊕ FourTempBytes[a]
    c = c + 1
```

Fig. 28 The KeyExpansion algorithm performs in place expansion of the key.

2.1.5. AddRoundKey Function.

With the key expansion complete, most of the core functions of Rijndael have been
implemented. The only core function that remains to be implemented deals with actually
utilizing the key to further *confuse* the input data and thus tie the cipher text to the user's
secret key. This function performs a simple XOR operation (addition) between the
intermediate *state* (block of data) being operated on and a part of the expanded key. The
key added with AddRoundKey (Figure 29) is dependent on the current *round* of
encryption that is being performed. With the 256-bit key expansion performed in this
implementation, 14 rounds of encryption are performed on each block, with each round
adding a different 32-byte round key from the expanded key.


```
set keyOffset = inputRound * 32
for i = 0 to 31
  state[i] = state[i] ⊕ inputKey[keyOffset]
  keyOffset = keyOffset + 1
```

Fig. 29 AddRoundKey modifies the state by XORing with a round key.

<u>2.1.6. Other Rijndael Implementation Details.</u>

With the core Rijndael functions finished, only a few more functions were required to have a working Rijndael implementation. The functions outlined here deal with background processes like, CBC mode, padding, as well as the basic doRounds and doInvRounds functions that use all core functions to perform the encryption.

<u>2.1.6.1. Implementing doRounds and doInvRounds Functions.</u>

The doRounds function (and inverse function) performs all the confusion and diffusion functions previously explained. The order in which these core functions are called by doRounds were carefully chosen by Rijndael's authors to maintain high security and secrecy. For decryption, the inverse of each core function is used in an opposite calling order. The doRounds function is exactly as shown in Figure 30, and directly operates on the current block of data (state). The doInvRounds function performs actions in the opposite order, with the opposite key order, and calls inverse functions to undo the encryption (Figure 31).

```
addRoundKey(0)
for i = 0 to Rounds – 1 (14 rounds for 256-bit blocks)
   SubstituteBytes()
   ShiftRows()
   MixColumns()
   addRoundKey(i)
SubstituteBytes()
ShiftRows()
addRoundKey(14)
```

Fig. 30 doRounds performs Rijndael encryption on the current block of data.

```
addRoundKey(14)
invShiftRows()
for i = Rounds – 1 downto 1
    invSubstituteBytes()
    addRoundKey(i)
    invMixColumns()
    invShiftRows()
invSubstituteBytes()
addRoundKey(0)
```

Fig. 31 doInvRounds performs Rijndael decryption on the current block of data.

2.1.6.2. Implementing CBC Mode.

CBC mode, outlined by Figure 9, requires the generation of an initialization

vector (IV). As mentioned previously, the IV does not need to be kept secret from the

attacker. In fact, it cannot be kept secret from the attacker as it is required to decrypt the

file properly. For this reason, the IV is typically stored as the first block in the cipher

text. As a rule, the final IV must be random and never re-used with the same key. This

unique IV can be achieved from non-uniform random input data by encrypting the IV

with the user's key. For platform independence, the implementation uses clock drift

calculations to generate the IV as shown in Figure 32:

```
array IV[32]
set clockTime = 0
for i = 0 to 31
    IV[i] = 0
    clockTime = clock()
    while(clockTime == clock())
        IV[i] = IV[i] + 1
```

Fig. 32 Utilizing clock drift for cross platform CBC IV generation.

Though the entropy of the clock drift IV calculation is dependent on the current

system usage, its output is typically not uniform random (Figure 33). After encryption

with the user's key, the output (as per typical Rijndael encryption) is uniform random and

42

appropriate for use as an IV. For extra mixing, a CBC style XOR is also performed

during IV generation (Figure 34).
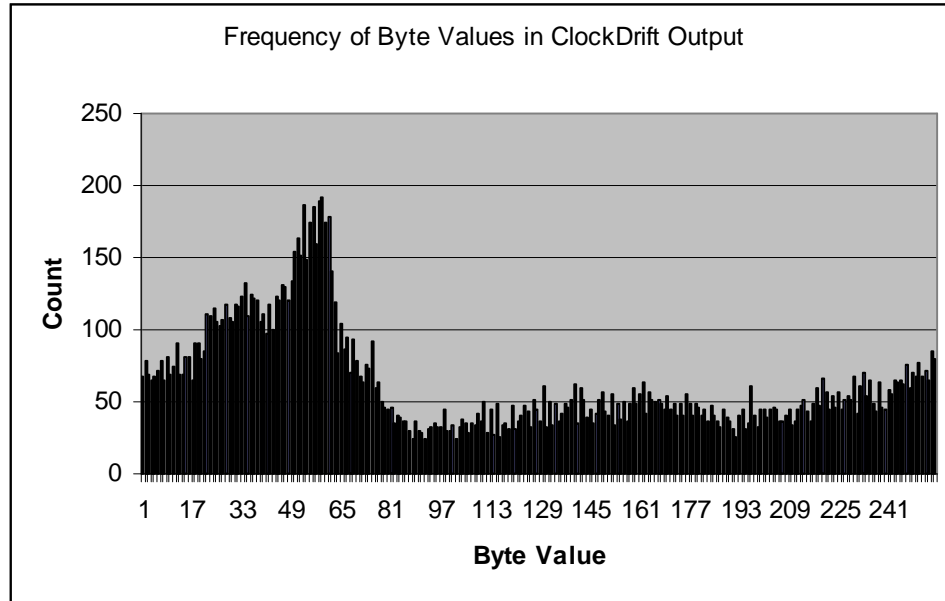
Fig 33. Frequency of Bytes in clock drift output.

```
array IV[32]
array Temp[32]

ClockDriftFill(IV)   //Fill IV with clock drift values
doRounds(IV)         //Do Encryption rounds on the IV
copy(IV, temp)        //Copy the IV array to the Temp array
ClockDriftFill(IV)  //Fill IV with clock drift values
XOR(IV, temp)     //XOR all 32 IV entries with the Temp entries
doRounds(IV)        //Do Encryption rounds on the IV

return IV          //IV is ready.
```

Fig. 34 InitIV function utilizes clock drift, encryption, and CBC to create IV.

With the IV generated, the basic functionality of CBC mode only requires two

trivial functions: one to update the current CBC state (a memcopy) and one to XOR each

entry of the CBC state with the encryption state. These simple functions were even used

in the InitIV function (Figure 34).

43

2.1.6.3. Implementing the Padding Functions.

The basics of the padding functions are trivial in nature but quite tedious in implementation. The general idea employed here is detailed in Figure 11. The idea is to pad the last block of data with a value equal to the number of padding bytes plus the size of a complete block. In addition, a complete block of padding is always encrypted as the final block; this method avoids the problem of removing padding bytes mentioned earlier. After padding and encrypting the final block, a second complete block of padding (with the same value) is encrypted and outputted. For decryption, the second to last block is decrypted, and then the last block's decrypted data is used to remove padding. Testing and perfecting the padding functions required a lot of trial and error, which seems typical regardless of the simplicity of the method.
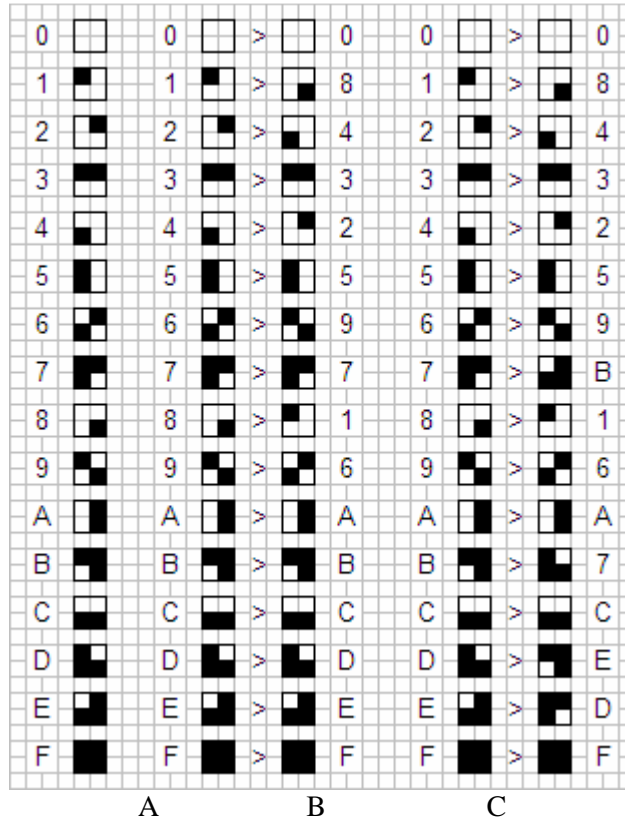
Figure 35. a. Possible Configurations
b. Billiard Ball Transition Rules
c. Bounce Gas Transition Rules

2.2. Implementing Margolus Automata

The original inspiration to use a Margolus style automata came from the undeniable reversibility of the standard billiard ball model (BBM) (Figure 35b) [6]. The BBM is a two state Margolus automata that simulates particle interactions as if they were bouncing balls. The BBM is interesting because its simple rules are reversible and can be run forward or backward in time. It was thought that if the BBM were tied to an S-box or bits of data as a diffusion step, it would be possible to use the BBM modify and restore the data. Reversibility of this kind is due to symmetry between transition rules and symmetrical states. With such a symmetrical Margolus neighborhood automata, reversing the calculation of generations only requires an inversion of the current

EvenOdd variable state. Though it seems nice, this type of reversibility turned out to be unnecessary for S-boxes because the S-box, automata generation, and current block of data are all dependent. This means that the block being processed and the S-box must be synchronized between encryption and decryption, not the reversed order.
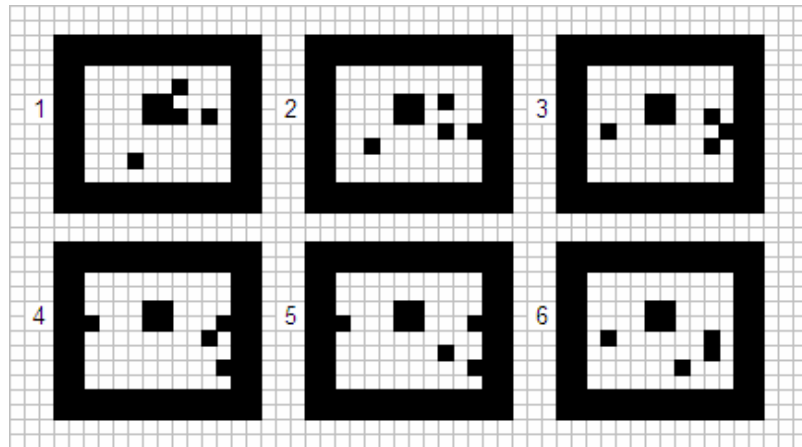
Figure 36 Billiard ball model interactions.

After taking a closer look at the mixing performance of the BBM, it was observed that some configurations remained static for each generation, namely 2x2 blocks of living cells (Figure 36) and any cluster of living cells. Having many configurations with null transition rules would limit the amount of scrambling performed; thus it was decided that automata rules that mix more uniformly would be preferable for this application.

Figure. 37 Weak Margolus rule showing clustering across generations.

Finding rules with certain properties can be difficult in some automata systems. This difficulty increases with the size of the neighborhood and number of possible states. In this system, there was the option of allowing the users to generate their own rules key. However, based on visual results of random rules, some patterns emerge that severely weaken the S-box's uniformity (Fig 37.). The rule chosen for the final design was the Bounce Gas rule invented by Tim Tyler [7] (Figure 35c). This rule is described by the author as a uniform gas rule, and based on experimental results, it does distribute 'on' and 'off' cells, or particles, uniformly (Figure 38). Due to symmetries in its transition rules (Figure 39), it also is reversible and thus applicable as a uniform bit-diffusion step in a different CAC design.

| Symmetrical starting pattern diffusion retains symmetry seen in rules. | Non-symmetrical initial pattern single cell added causes eventual uniform diffusion of cells. |
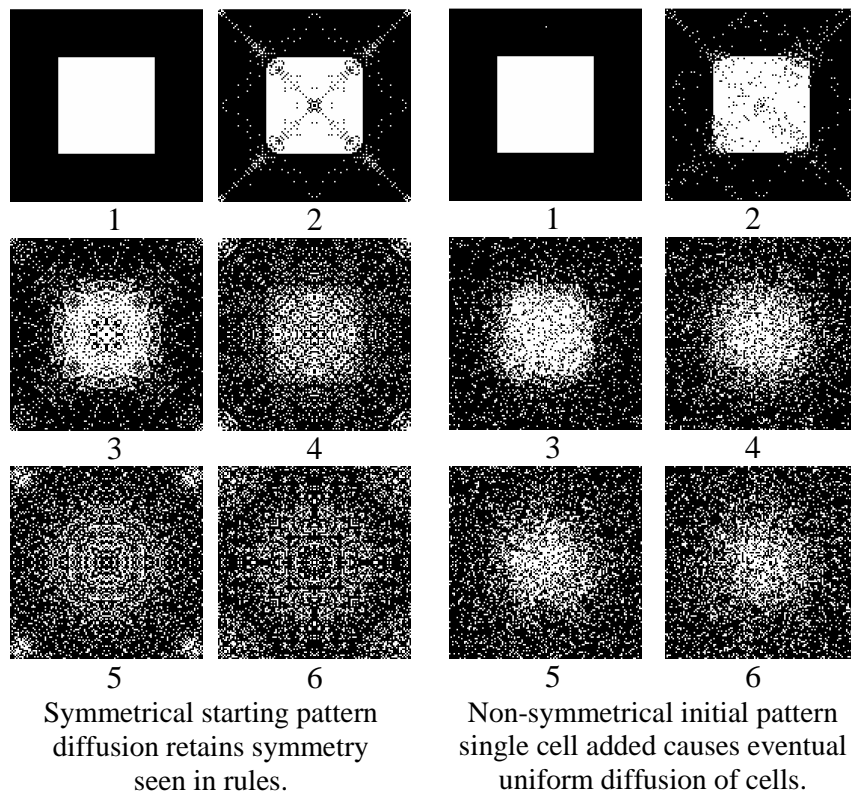
Fig. 38. Visualization of uniform distribution of on and off cells under the uniform bounce gas rule in 100 generation steps.

The main advantage of the Margolus neighborhood for S-box modification is the simple 2x2-block design. This partitioning scheme allowed all rules to be simple swaps between horizontal, vertical, or diagonal pairs of cells. For each of the 16 configurations possible with 2x2 blocks, any number of the six swaps may be performed.

Figure 39 Symmetry in BounceGas Rules allowing full reversibility, uniform dispersion, and symmetrical pattern propagation.

Though the implementation only uses the BounceGas rule, having a swapping design allows all possible rules, weak or not, to be used for S-box modification, as losing data is impossible. Of course, due to this feature, some transitions possible in a normal Margolus implementations are impossible, a transition from state zero to fifteen for example. There was also the inverse S-box to swap, but maintaining invertibility between the S-box and Inverse S-box was a trivial operation. The order in which the swaps are processed is fixed and may affect the results.

Figure 40. S-Box viewed as a CA grid with threshold 0x7f,
1:1 alive to dead ratio.

Another useful feature of the design is the threshold value. This value is used to determine which cells are considered *live* and which are considered *dead,* state 1 and 0 respectively. The default value is '127' for uniformity purposes. With this setting, half the cells in the S-box will be *live* and half will be *dead*. Figure 40 shows how a typical S-Box looks when this threshold value is applied. Slight modification of the threshold value drastically alters the results, hence its use as a user parameter. As the threshold varies further from the half-way value of 127, fewer swaps will be performed. When the threshold value reaches a value of 0 or 255, no swaps will be performed. No swaps are performed when a partition is in configuration 0 or F, as per the BounceGas rules shown in Figure 35c. Thus, the threshold values are limited to a range to ensure the ratio of live to dead cells is not too skewed, which would hinder diffusion.

2.2.1. Determining the Rule Structure.

The 16*6 possible swapping rules available for defining the transition function were designed as a 16x6 array of Booleans. For each of the 16 configurations, or states

49

as it is used here, there are six Boolean values indicating whether to perform that swap. This method can easily perform the swaps on the cell with the algorithm shown in Figure 41:

```
for swapType = 0 to 5
  if(rulesMatrix[currentState][swapType] == true)
  switch(swapType)
    case 0
      swapDiag1
    case 1
      swapDiag2
    …
    case 5
      swapVert2
```
Figure 41. DoTransition function based on 'currentState'

2.2.2. Determining the Configuration.

Determining the configuration of a 4x4 block is performed very often and therefore needs to be fast. The way the states are numbered in Figure 35a follows a binary pattern. This numbering scheme coupled with the rule matrix implemented, allows the current configuration value to be calculated very efficiently.   In the implementation it is calculated with a single line define statement, encompassing the general formula shown in Figure 42 which sums up the decimal value of the four 'bits' in the configuration. This gives the state number.

```
set currentState = 0
currentState = currentState + getCellValue(x+1,y+1) > threshHold ? 8 : 0
currentState = currentState + getCellValue(x,y+1) > threshHold ? 4 : 0
currentState = currentState + getCellValue(x+1,y) > threshHold ? 2 : 0
currentState = currentState + getCellValue(x,y) > threshHold ? 1 : 0
```

Figure 42. GetConfiguration function, where "?:" represents the C/C++ tertiary operator and threshHold is an 8-bit value with default 127.

With the simple algorithms for applying the swaps and for determining the current configuration, there comes an equally simple method for making it all work. As shown previously in Figure 2c., the alternating even and odd partitioning scheme inherent in the Margolus neighborhood creates an environment where living cells, or particles, are constantly moving. The partitioning scheme can easily be handled by processing the map starting from position (0,0) or position (1,1), based on the even or odd state. Border handling is another concern for partitioning, and naturally wrapping the borders into a torus was preferred for this software implementation. The complete procedure for processing a generation of an XxY map is quite simple as Margolus intended (Figure 43). The S-box is simply an array; thus, the CA was easily programmed to work within one-dimensional arrays by using the basic formula: offset = x + y*width.

```
evenOdd = !evenOdd
for x = evenOdd to X-1
  for y = evenOdd to Y-1
     doTransition(x, y, getConfiguration(x, y))
     y = y + 2
  x = x + 2
```

Figure. 43 Processing the map where doTransition and getConfiguration are defined by Figures 41 and 42 respectively.

2.3. CAC Construction.

Combining the cellular automata to the crypto system was painless with this solution. Once the automata is initialized, its operation only requires per block calls to perform generations on the S-boxes. The addition of the CA to the encryption process yields the flowchart in Figure 44, and Appendix A6 provides a more detailed flowchart. Figure 44, displays the process of encrypting an entire file, where CBC, padding, and CA

generations are all performed. At this point, due to it being a trivial modification of the

CA source class, a key modification automata with the same rules was also added. This

addition alone has the potential of increasing security on many ciphers.
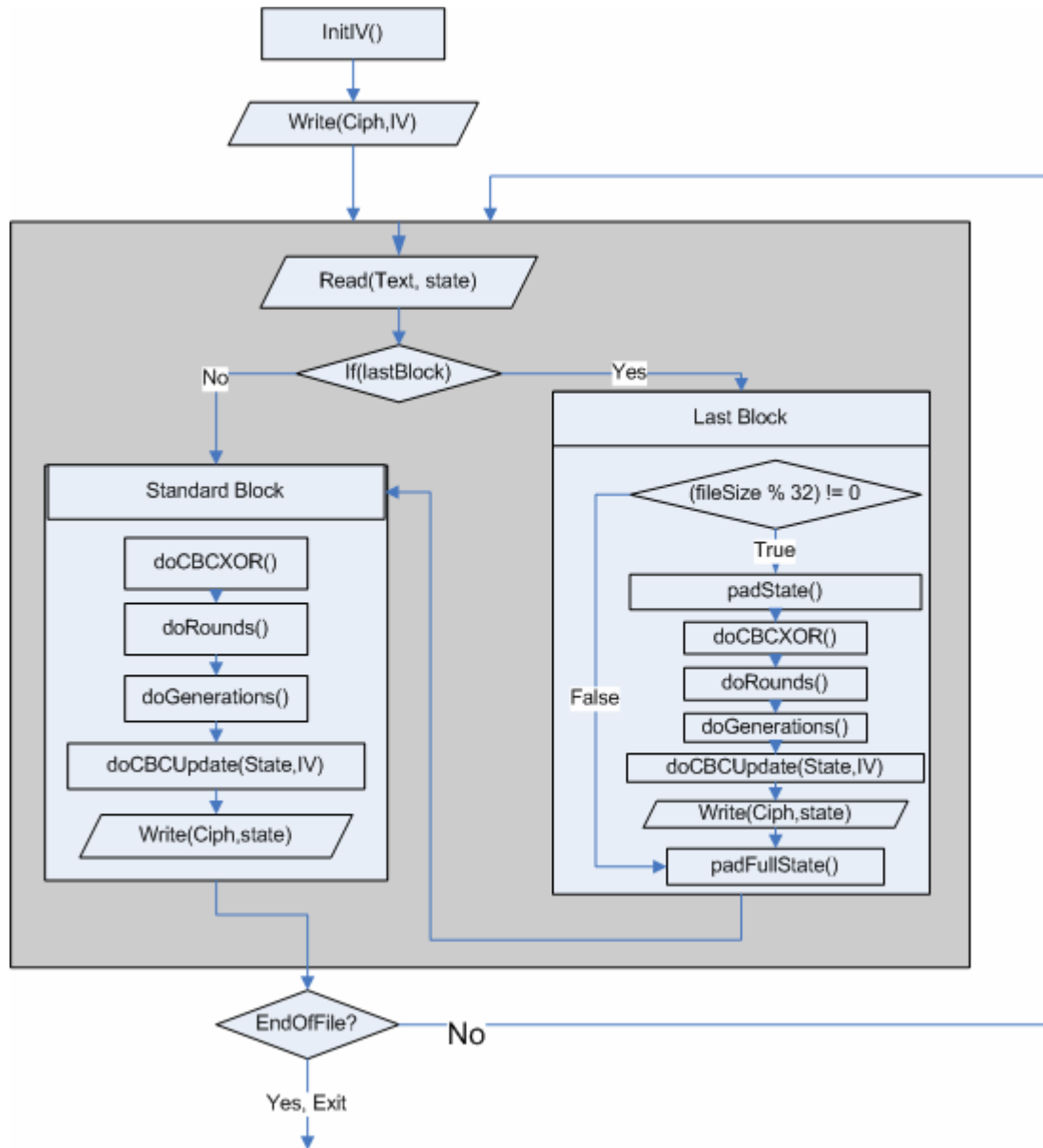
Figure 44. Flow chart showing the process of encrypting a
file where doRounds does the operations performed in Figure 13.

3. ANALYSIS OF CRYPTOSYSTEM

During the implementation of the CAC, the modular systematic approach was used to verify the proper operation of each algorithm. With the implementation complete and working, analysis of the results needs to be performed. The hypothesis of this analysis is that the CAC designed will offer improvements to the S-box cryptographic primitive with a minimal performance impact. The analysis of security focuses on the differences between Rijndael and the CAC. The security of the specific binary implementation of the CAC is not a major focus, as minor coding mistakes and insecure user input methods do not reflect on the security of the actual design.

3.1. Analyzing the Cellular Automata.

The Margolus automata was analyzed in a number of ways to ensure it meets the criteria for which it was chosen. Tim Tyler, the author of the BounceGas rule [7], claims that the rule simulates a "uniform gas". This claim, if true, benefits the CAC by ensuring no bias or clustering of values in the S-box. If the initial S-box is sufficiently random, then a uniform gas rule strongly suggests the creation of equally random S-boxes. To verify this uniformity claim, specifically for the 2-state automata, a number of tests were designed and performed.

3.1.1. Distribution of States.

In the most basic sense, all Margolus automata rules result in 1-bit bitmaps. The S-box also must be viewed like this, which is why the threshold value is required.

Looking at the S-box as a 1-bit bitmap, it was thought that if the BounceGas rule is in fact uniform, then the average mid-point for the ones and the zeros should be roughly equal and near the center of the map at all times. A simple experiment was performed to test this idea. The experiment required starting with a non-uniform map with roughly one-half all 'ones' and the other half all 'zeros'. With the BounceGas rules, a symmetrical starting state will always remain symmetrical; thus, a few random cells on either side were inverted from the start. After the map was initialized, single generations were performed (Figure 45). Based on the visualization of the data, it is obvious that the non-uniform starting state quickly becomes uniformly mixed and remains that way.
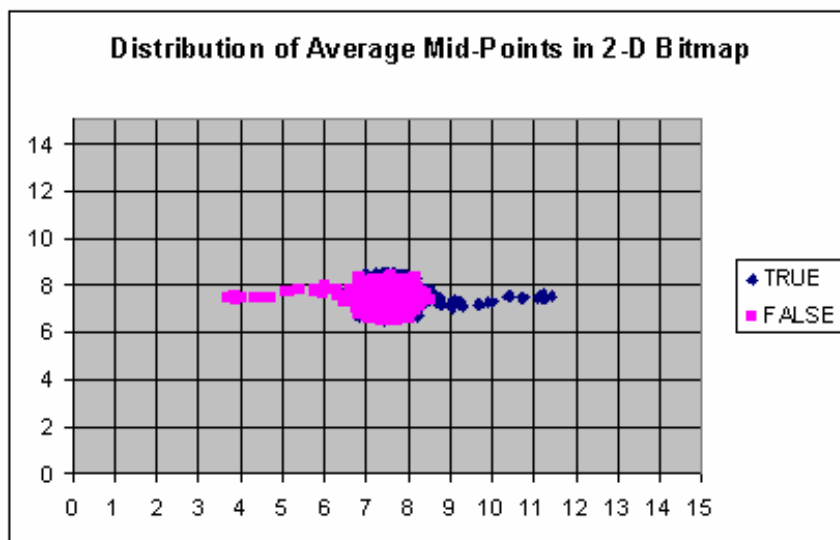
Figure. 45 Distribution of average Mid-Points for both states, from a non-uniform initial state in a 16x16 bitmap. 5782 trials.

After each generation the automata modified S-box, was analyzed to find the average location of the two states. This means, for each S-box entry, the threshold was applied to determine if the entry is considered true or false (on or off, alive or dead, etc). Based on this state, the coordinates for that entry are added to one or the other running average. These averages represent the overall location of the two possible states. If the

two states are equally spread among the map, then they will both have values near the center of the map (7,7). This would also suggest that from a non-uniform starting state the distance of the midpoints from the center of the map (7,7) for both 'true' and 'false' states would approach zero, after some number of generations (Figure 46). Due to the non-uniform starting states, the distance from center begins large but decreases each generation for both data sets. The data also shows bias towards one set over the other because the number of true states is not equal to the number of false states. When the same tests are performed on a uniform initial S-box, like those generated in the implementation, the S-boxes bitmaps remain uniform.
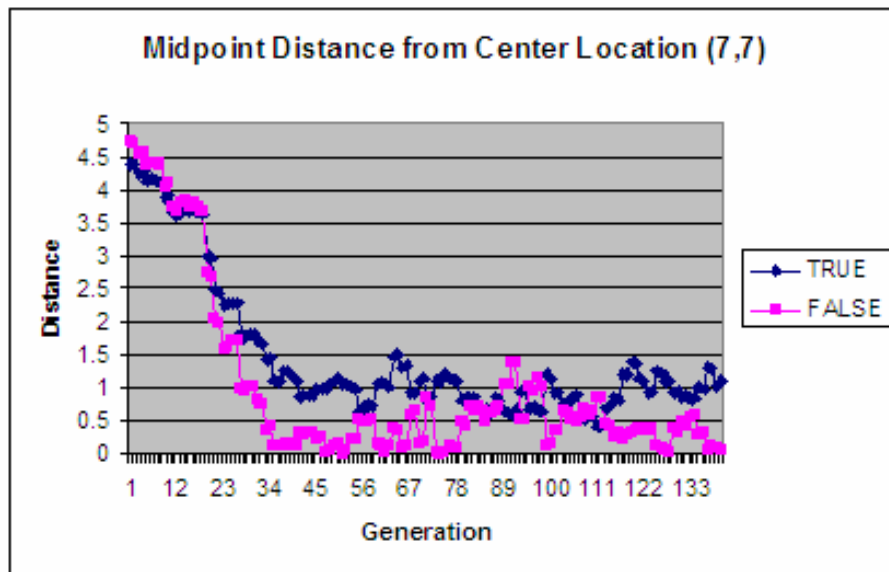
Figure. 46 Midpoint distance from center location, from a non-uniform initial state. 140 of 5143 trials shown.

3.1.2. Distribution of Swap Counts.

The operation of the Margolus automata was investigated in terms of the number of swaps performed per generation. It was expected that the data would be normally distributed. Any results other than normal would not fit the operation of a uniform and chaotic system. Figure 47 indicates that the distribution is normal with a large variance

and an average of 48 swaps per generation under the BounceGas rules. Theoretically, it may be possible for an attacker to determine the user's initial S-box based on the deterministic swap counts, though the availability of 128 threshold values and 128 initial S-boxes makes this unlikely. Even then, the CAC design is as strong as Rijndael.
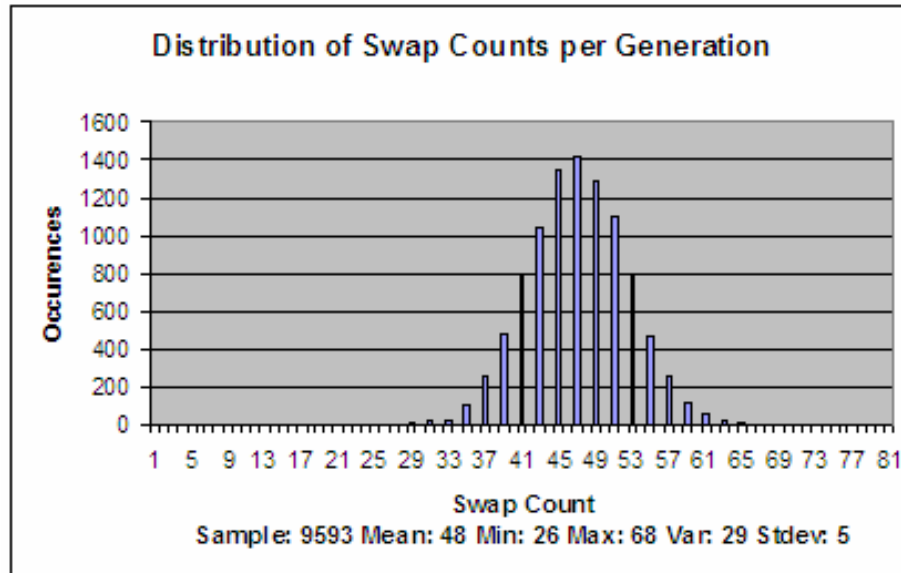
Figure 47. BounceGas rules swap count per generation with threshold at 127.

3.2. Analyzing the Generated S-boxes.

S-boxes, a major part of the confusion step of the cipher, require some properties to avoid leaking information into the cipher text. The S-boxes initially generated with the Galois generators are said – with some arguments [20] [21] – to be strong in some properties like the strict avalanche criterion and non-linearity [9]. Based on the previous analysis of the CA uniform dispersion properties, and thus randomness in byte values, it was expected that the generated S-boxes would also perform well in the same tests. One direct result of using a new S-box every block is the effect it has on the data histogram of a file. If a large file of English text were simply substituted with the static S-box values,

frequency analysis could easily be used to decrypt the cipher text. However, if the same file data were substituted with different values for each 32-byte block, this attack would be difficult due to the limited sample size of 32 bytes for frequency analysis. Some of these tests could possibly be redundant because a new S-box is used for every block; however, a single weak S-box could theoretically make the CAC weaker than Rijndael alone.

3.2.1. Bit Change and Avalanche Criteria.

S-boxes are typically compared with various avalanche criteria. In his paper "Avalanche and Bit Independence Properties for the Ensembles of Randomly Chosen nxn S-Boxes" [29], Isil Virgili found that it is unrealistic to expect most S-boxes to meet the strict avalanche criterion (SAC) and more realistic to accept S-boxes within an error range. This interpretation of the SAC and CAC led to the design of two simpler tests strictly for S-box avalanche testing, as opposed to entire cryptosystem testing. To meet the criteria of this interpretation of SAC and CAC based on Isil Virgili's research, the numbers should always indicate about 50% bit changes in the results.

The first test counts the number of bit changes for all 256 input and output byte combinations in the generated S-box. Figure 48 shows the results of the first tests when comparing the 128 possible generated S-boxes and 256 dynamic S-boxes. Based on the limited data for Static S-boxes, the results suggest that the dynamic S-boxes perform nearly as well with just a little more variance.
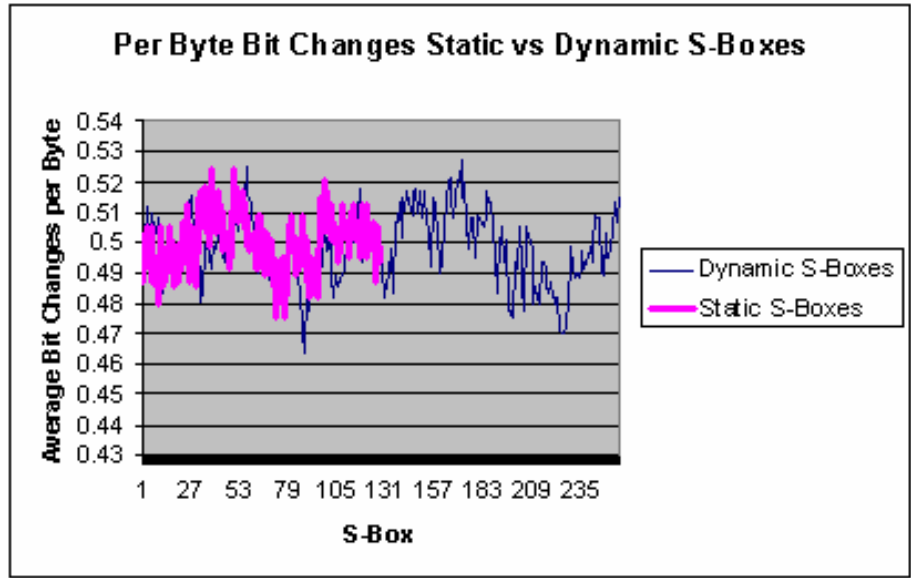
Figure 48. Average per byte bit changes in static and dynamic S-boxes.

The second test counts the number of bit changes between each input byte and its outputs when each bit is flipped. Figure 49 shows a comparison between the 128 static and 256 dynamic S-boxes. This test, while closely related to the first, is closer to a second order bit-independence test. Interestingly enough, the Galois generated S-boxes have a constant avalanche of 0.504883, which is due to their 9-term arithmetic construction [20]. The dynamic S-boxes perform nearly as well, with a similar mean and slightly more variance.
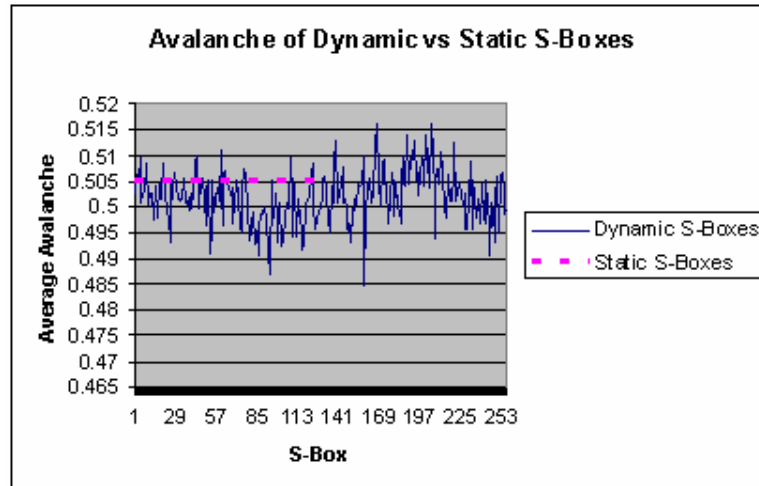
Figure 49. Comparison of dynamic and static S-boxes avalanche.

3.2.2. Non-Linearity Measures.

Non-linearity is a typical measure of S-boxes, and the S-boxes generated by Rijndael are said to have [9] "optimum worst-case non-linearity properties". Non-linearity seems like a good metric for the comparisons of dynamic and static S-boxes. Typically, accidental affine operations are often weaknesses in new cryptosystems. Non-linearity can be thought of as the absence of that type of weakness. It is basically a measure of the number of bits that must change in the truth table of a Boolean function to reach the closest affine function [30]. According to Terry Ritter [31], non-linearity is measured by forming the 1-bit wide truth tables for each output bit and then performing a Fast Walsh-Hadamard transform (FWT) on the truth table to find the correlation count between the truth table and the set of affine functions. The functions for measuring non-linearity were ported to C++ from Terry Ritter's JavaScript FWT and non-linearity testing sources with the authors permission. The FWT calculates the difference away from the affine functions, and the largest distance possible from any affine function is plus or minus one half of the bits, or 128 in this 8-bit case.

That said, the reported minimum non-linearity for an S-Box is calculated as $2^n/2 - abs(maxDist)$, where n is the number of bits. Though each entry in the S-box will have a non-linearity measure, only the minimum is reported for a worst-case comparison. Based on the results of this test, it is interesting to note that all 128 of the statically generated S-boxes have the same minimum non-linearity (minNL = 112). This is a direct result of the way the S-box is generated by its 9-term algebraic expression [20]. Figure 50 shows the resulting minimum non-linearity measures on some dynamically generated S-boxes. A value of 128 is the unreachable best-case upper bound of non-linearity. While the 128 static S-boxes perform very well at 112, the dynamic S-boxes perform decently with an average minimum non-linearity of 99 based on 256 trials.
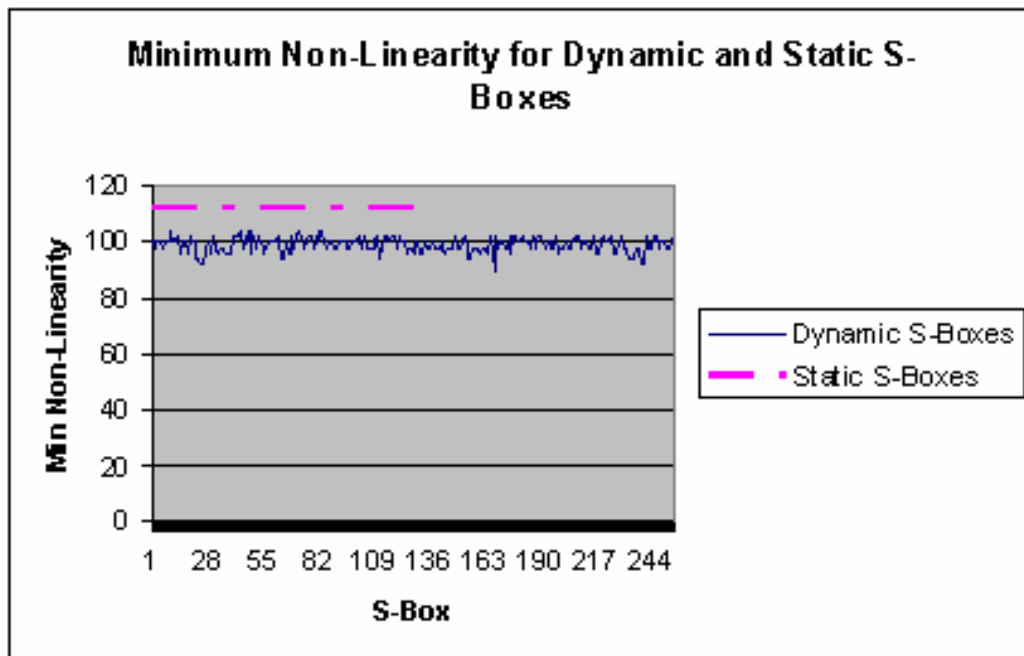
Figure 50. Comparison of minimum non-linearity for dynamic and static S-boxes.

Based on research by Terry Ritter [30] (Figure 51), the distribution of minimum non-linearity for random 8-bit S-boxes has an average of 100, which is supported by the experimental results of the dynamic S-boxes (Figure 52). One of the original indicators

showing the importance of the uniform dispersion properties of the BounceGas rule comes from comparing the distribution of MinNL under randomly chosen Margolus rules.

```
0.35  |                              *
0.3   |                              *
0.25  |                      *       *
0.2   |                      *   *   *
0.15  |                      *   *   *
0.1   |                  *   *   *   *
0.05  |              *   *   *   *   *
0.00  |          *   *   *   *   *   *   *
Prob  +--+--+--+--+--+--+--+--
          92      96      100     104
                  Nonlinearity
```
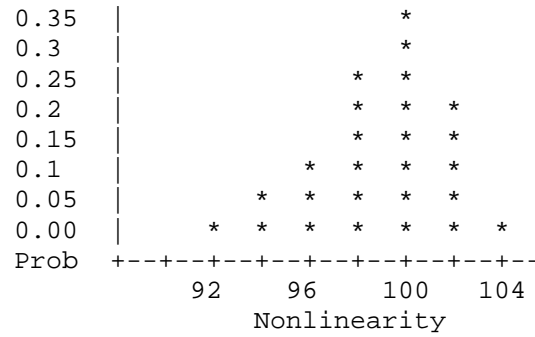
Figure 51. Nonlinearity Distribution in random 8-Bit Tables as reported in Terry Ritters Research. [30][31]
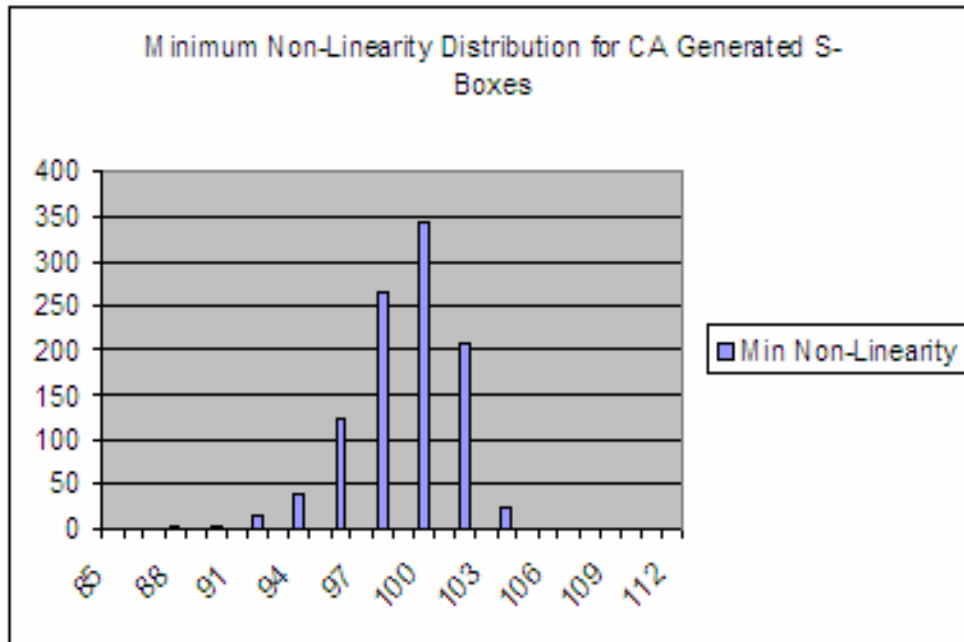
Figure 52. Experimental results showing distribution of MinNL for CA generated S-boxes under the BounceGas rule with threshold 127. 500 trials.
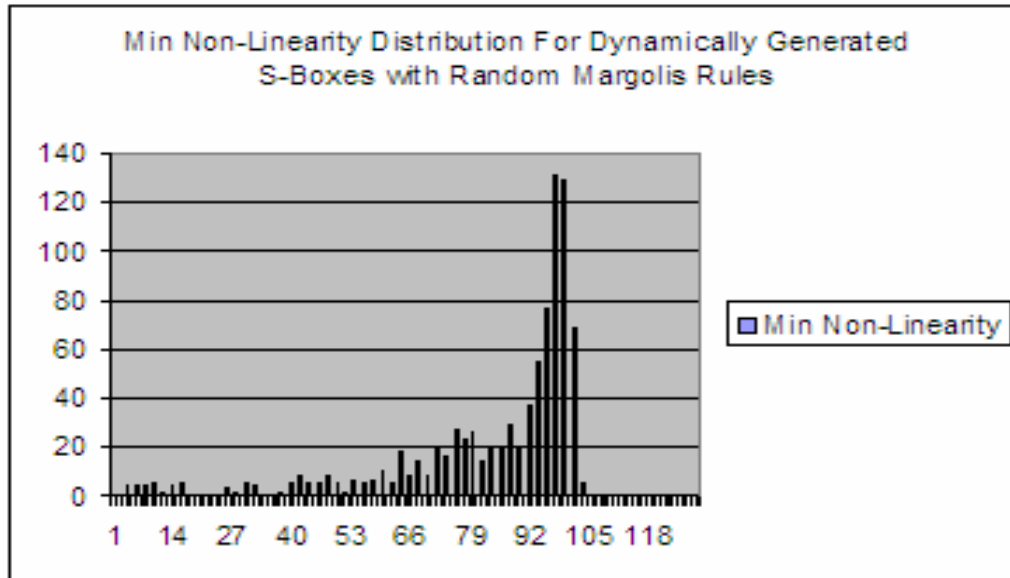
Figure 53. Distribution of MinNL for CA generated S-boxes under random rules with threshold 127. 500 Trials

The results (Figure 53) of this test clearly shows that some Margolus rules can lead to very linear S-boxes, which would be useless from a security standpoint. The test modified the S-box based on a random rule for 50 generations. It then output the minumum non-linearity on the CA modified S-box This was the test that showed the importance of having good CA rules (like BounceGas) for the Margolus CA transition function.

Though the BounceGas rule performs decently well at generating non-linear S-boxes, it still has a lower non-linearity than the specially crafted Galois generated static S-boxes which had non-linearity 112. Terry Ritter's research [30] on random 4-bit permutation S-boxes for DES-like Feistel ciphers, shows that randomizing the S-boxes is a very bad idea. He does conclude, however, that random 8-bit permutation S-boxes are very rarely weak, and the bad stigma about randomized S-boxes is most likely for the special case of DES or 4-bit S-boxes. That said, the CA generated S-boxes with the average MinNL of 99 appear to be acceptable.

3.3. Analyzing the CAC Output and Performance.

The overall performance of the resulting CAC can be analyzed and compared to Rijndael in many ways. If the CAC design is not any weaker than the Rijndael design, then the results should be very similar in both cases.

3.3.1. Maurer's Universal Statistical Test.

Maurer's universal statistical test is typically used to measure the performance of random bit generators [32]. It reports a value representing how well the bit stream could be compressed by looking for patterns in the input. With enough input data, Maurer's test is said to be universal in that it can be performed instead of the five common tests: mono-bit test, two-bit test, runs test, poker test, and the auto-correlation test [32]. If we assume any well-encrypted cipher text will look like random data, then this test can be applied to compare cipher text outputs. The algorithm (Figure 54.) [32] reports a value of about 7.18 for true uniform random 8-bit data streams, with lower values showing less random data. An internal parameter to the statistic is the number of bits in each entry, represented by L. Computing the test statistic requires a large amount of data, or roughly $1000 * 2^L$ L-bit blocks. Q represents the number of blocks used to initialize the 2^L table entries, and should be of size at least $10*2^L$. K represents the other $990 * 2^L$ blocks to be processed, and A_i represents the actual L-bit entries from the random source.

$$X_u = \frac{1}{K} \sum_{i=Q+1}^{Q+K} \lg A_i.$$

Figure 54. Computing Maurer's Test Statistic [32]

The results of Maurer's Universal Statistical test (with L=8) on the cipher text outputs with various encryption parameters show uniform random outputs across the board (Table 4), which implies that the CAC is not weaker than Rijndael alone.

	Generations	Maurer's
Moby-Dick.txt	0	3.8891
Ciph-Rijndael	0	7.1848
Ciph-CAC	1	7.1878
Ciph-CAC	2	7.1824
Ciph-CAC	3	7.1809
Ciph-CAC	4	7.1867
Ciph-CAC	5	7.1868
Ciph-CAC	10	7.1831
UniformRandom	0	7.1850

Figure 55. Maurer's experimental results comparing CAC outputs to english text and random uniform data.

3.3.2. Entropy and Conditional Entropy.

Shannon entropy, or information entropy, is a measure of the amount of information contained in a random variable [33].   A constant pattern has zero entropy, indicating that zero bits of data are required to transfer such a message. English text typically has entropy of about 1.5 bits per letter, while a truly random string of characters would have entropy of 8.0. The standard entropy formula [34] (Figure 56a) provides nearly the same information the Maurer's test reports, but the conditional entropy (Figure 56b) [35] formula is often used as a measure of secrecy. Conditional entropy provides a measure of the similarity between two discrete random variables, which can be used to determine whether the two variables are independent or to what extent they are dependent on each other. The conditional entropy between a plaintext and its cipher text would report a value of zero if the cipher text were to provide all the information necessary to undo the encryption, as is the case with the simple shift affine cipher. A cryptosystem is said to have complete secrecy if the conditional entropy between the texts and the entropy

of the original file are equal, which implies that a one-time pad (OTP) is the only perfectly secret cipher.

$$H(X) = E(I(X)) = \sum_{i=1}^{n} p(x_i) \log_2 (1/p(x_i))$$

$$= -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

a)

$$H(Y|X) \stackrel{\text{def}}{=} \sum_{x \in \mathcal{X}} p(x) H(Y|X = x)$$

$$= -\sum_{x \in \mathcal{X}} p(x) \sum_{y \in \mathcal{Y}} p(y|x) \log p(y|x)$$

$$= -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(y|x)$$

$$= -E_{p(x,y)} \log p(Y|X).$$

b)

Figure 56. Definition of entropy
a) H(X) and conditional entropy [34]
b) H(Y|X) [35]

The entropy and conditional entropy were used compare cipher texts produced by Rijndael, the CAC, and an OTP cipher. To provide a more clear comparison between the systems, the input data was aligned along a block boundary and all CBC and padding code was removed. The OTP cipher was simulated by generating a random block of data for the cipher text. The results (Table 4) indicate that the CAC is no weaker than Rijndael and that both perform nearly as well as the OTP, which was somewhat unexpected. The results also suggest that increasing the number of CA generations does not have much effect on the results. Thus as far as conditional entropy and secrecy are concerned, the CAC and Rijndael perform extremely well, and the differences between them are statistically insignificant.

X	Y	Entropy X	Entropy Y	Cond Entropy	Diff
magna-carta.dat	magna-carta-fake-OTP.dat	4.4147	7.99284	4.18532	0.22938
magna-carta.dat	magna-carta-ciph-Rijndael.dat	4.4147	7.99436	4.06934	0.34536
magna-carta.dat	magna-carta-ciph-CA1.dat	4.4147	7.99384	4.12997	0.28473
magna-carta.dat	magna-carta-ciph-CA2.dat	4.4147	7.99345	4.06829	0.34641
magna-carta.dat	magna-carta-ciph-CA3.dat	4.4147	7.99404	4.10717	0.30753
magna-carta.dat	magna-carta-ciph-CA4.dat	4.4147	7.99505	4.14187	0.27283
magna-carta.dat	magna-carta-ciph-CA5.dat	4.4147	7.99389	4.07648	0.33822
magna-carta.dat	magna-carta-ciph-CA10.dat	4.4147	7.9943	4.15784	0.25686
magna-carta.dat	magna-carta.dat	4.4147	4.4147	0	4.4147

X	Y	Entropy X	Entropy Y	Cond Entropy	Diff
moby-dick.dat	moby-dick-fake-OTP.dat	4.49714	7.99972	4.48107	0.01607
moby-dick.dat	moby-dick-ciph-Rijndael.dat	4.49714	7.99973	4.47943	0.01771
moby-dick.dat	moby-dick-ciph-CA1.dat	4.49714	7.99973	4.46895	0.02819
moby-dick.dat	moby-dick-ciph-CA2.dat	4.49714	7.99973	4.46718	0.02996
moby-dick.dat	moby-dick-ciph-CA3.dat	4.49714	7.99969	4.47666	0.02048
moby-dick.dat	moby-dick-ciph-CA4.dat	4.49714	7.99974	4.47919	0.01795
moby-dick.dat	moby-dick-ciph-CA5.dat	4.49714	7.99974	4.47805	0.01909
moby-dick.dat	moby-dick-ciph-CA10.dat	4.49714	7.99972	4.46592	0.03122
moby-dick.dat	moby-dick.dat	4.49714	4.49714	0	4.49714

Table 4. Experimental Conditional Entropy Results

3.3.3. Data Histogram Results.

As mentioned previously, the simplest and most naïve method of judging the strength of a cipher is the data histogram approach. Any cipher with a chance of being better than the weakest affine shift ciphers will have a uniform data histogram for all its cipher text output. Statistical analysis of the cipher text output distributions would merely repeat the results of the entropy and Maurer's tests, therefore only a simple graphical check was performed (Figure 57). It is important to point out that all algorithms that utilize an S-box – ignoring their diffusion steps – are all nearly as weak as shift ciphers. Simply performing the same substitutions for each byte of the file allows the input and output frequencies to be directly compared. While this is only a weakness in a cipher that doesn't have a diffusion step, each block will have a unique histogram if this CAC had no diffusion step. This fact reduces the amount of data that can be collected for frequency attacks to the size of a single block, as opposed to the entire file.
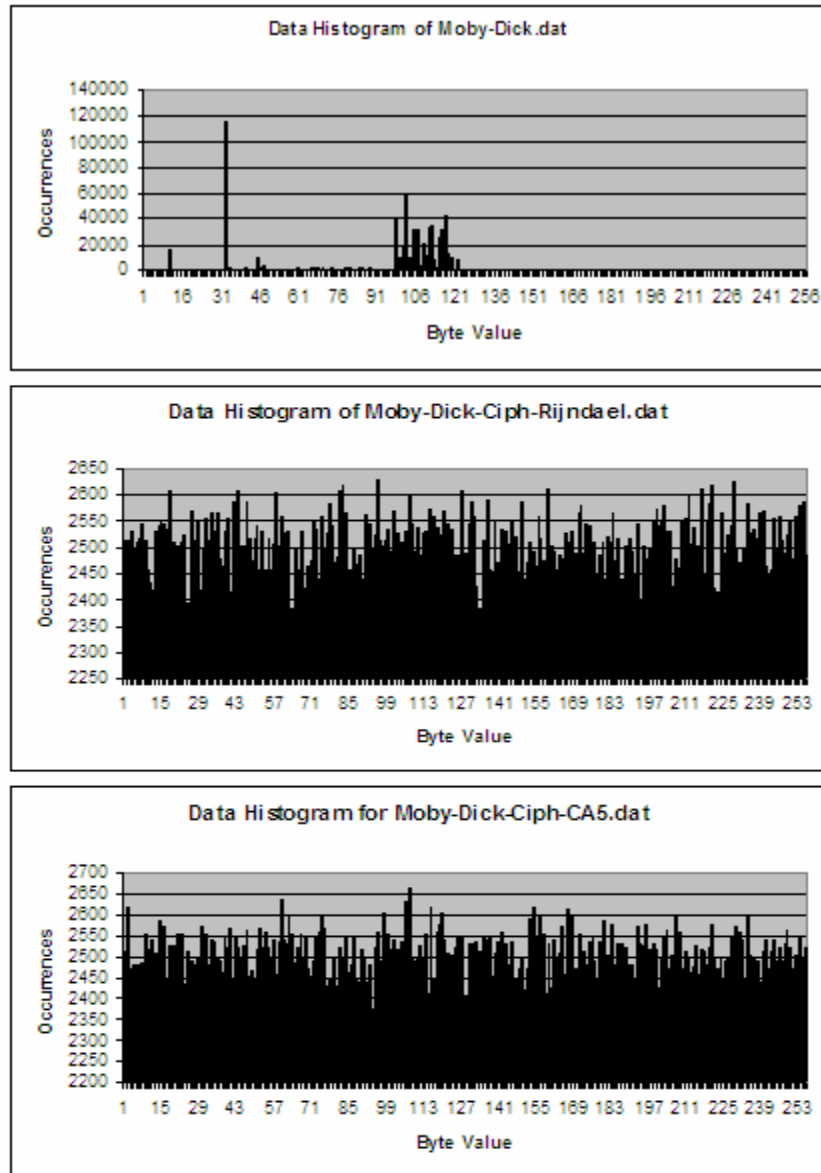
Figure 57. Comparison of original and output data histograms.
The two ciphertext histograms appear equally uniform.

3.3.4. Compression Results.

The compression test is a simple test that should support the conclusions of the
Maurer's and entropy tests. Based on the entropy of around 7.99 for all of the cipher
texts, the compressed file should be larger than the uncompressed file. If the
compression of a cipher text ever reduces the file size, the cipher or implementation is

flawed (as in ECB mode). Based on the results of compressing with BZIP2 (Table 5), the

CAC passes this simple compression test.

File Name	Uncompressed Size	Compressed Size	Diff
magna-carta.dat	29824	9638	-20186
magna-carta-ciph-rijndael.dat	29824	30304	480
magna-carta-ciph-ca1.dat	29824	30302	478
magna-carta-fake-OTP.dat	29824	30319	495
moby-dick.dat	643232	200486	-442746
moby-dick-ciph-rijndael.dat	643232	646597	3365
moby-dick-ciph-ca1.dat	643232	646299	3067
moby-dick-fake-OTP.dat	643232	646571	3339

Table 5. Experimental Results of BZIP2 compression test.

3.3.5. Time Analysis and Profiling Results.

Based on the analysis done thus far, the cryptosystem is not weaker than Rijndael.

If the perceived weaknesses in the standard static S-box approach ever lead toward a

cryptanalysis technique for breaking Rijndael, then the CAC design will be the more

secure algorithm. If this were to happen, there is still the question of whether or not the

extra processing required to dynamically modify the S-box in this way is worth the added

security. For this, a profile of the code while encrypting a file was performed (Table 6).

	InitIV	InitCipher	Encrypt	Decrypt	E - D Time
NoCA	0.638239	0.000087	11.763086	11.281403	0.481683
CA	0.640645	0.000088	17.366768	16.731485	0.635283
		CA - NoCA Time	5.603682	5.450082	
NoCA		Bytes/sec	356564.94	371789.22	
CA		Bytes/sec	241513.22	250683.31	
		Reduction %	32.266695	32.573809	

Table 6. Temporal differences between static and dynamic
S-boxes while encrypting a 4mb file with one generation/block.

These timings were taken to microsecond accuracy and the offer some startling

results. The processing of the simple CA code with one generation per block adds quite a

bit of overhead when encrypting a 4MB file. A 32% reduction in throughput was

unexpected. There are a lot of factors causing these performance issues. With a 4MB

file, there are 131,072 256-bit blocks, which corresponds directly to the number of generations performed. The overhead of generating a new S-box for each block of data, was much larger than expected.

A simple modification to the design would provide another user parameter to control the number of blocks processed between CA S-box modifications. Another factor is the large number of swaps being performed by the BounceGas rule. As shown in Figure 47, there is an average of 48 swaps performed per generation with the default threshold value of 127. Retaining synchronization between the S-Box and Inverse S-Box doubles this to an average of 96 swaps per generation. With a 4MB file with 131,072 blocks, the algorithm would perform 12,582,912 swaps on average with the default threshold of 127, meaning 12 million function calls. However, if the user modifies this threshold away from the default mid-way point of 127, the number of swaps per generation will always be reduced (on average).

The time analysis also points out the existence of a hypothetical problem somewhere in the implementation. The difference in encryption and decryption time is a commonly warned against problem. If an attacker were able to perform similar profiling or timing analysis, they may be able to exploit this information to gain limited knowledge of the plaintext or internal state. While this highly unlikely situation is mostly a hardware issue, timing attacks should not be ignored. In theory, each encryption and decryption function should use the same number of cpu cycles.

Based on these results and the typical non-linearity measures of random 8-bit S-boxes (Figure 51) [30], one idea stands out. If a future novel cryptosystem were to use dynamic S-boxes, it may be more efficient to just perform random permutations on the

S-box. It may be even more efficient to do this only once per file based on some parameters.

4. CONCLUSIONS

The Cellular Automata Cryptosystem designed for this thesis has been shown to operate as it was intended. The analysis performed indicates that the CAC design is not weaker than the original Rijndael algorithm. While the *implementation* may not be perfectly secure, as it wasn't a major focus, it does offer enhancements over numerous currently deployed AES/Rijndael products. This is based on the implementations' dynamic generation of the initial S-Box, which partially defeats the S-Box blanking attacks described by Kerrins and Kursawe [22]. It is important to note, however, that the implementation requires static table of the Galois generators to be stored in the binary. Though there is a minor checksum and verification performed on this table at program load, truly protecting the implementation from S-Box blanking requires further operating system level protections as Kerrins suggested. Thus the design only partially protects against S-Box blanking.

On the subject of the possible weaknesses of Rijndael's S-Box [20] [21], specifically the "9-term algebraic expression" mentioned by Jingmei and Fuller, the design fully protects against this. If the algebraic construction of Rijndael's S-Box was ever used to successfully attack AES/Rijndael, or was discovered to be an advantage for the NSA, a cryptosystem that implements dynamic S-boxes will undoubtedly be preferred. This of course assumes that the CA generated S-boxes very rarely have an

algebraic expression at all. The results of the Minimum Non-Linearity tests, wherein the CA generated S-boxes had an average of 99 and the Rijndael S-Boxes had a static value of 112, may raise some questions about the strength of CA S-boxes even though research suggests it is a non-issue for 8-bit S-boxes [31].

All data suggests that this system is far more resistant to cryptanalysis than the already "unbreakable" 256-bit Rijndael implementation with a single S-Box. However, it should be noted that choice of the standard S-box was chosen for programmatic performance tricks. This design removes the ability to cut corners on performance, and also adds significant overhead. The 32% reduction in encryption and decryption speed, with only a single generation per block, was quite unexpected. In its current form, the Margolus automata rules add too much overhead to the process, especially since a main idea behind this thesis was that CA are very fast and Margolus Automata are the fastest of the fast. This was a disappointing realization. Furthermore, taking advantage of the blistering speeds obtained using Cellular Automata in hardware ASICs, would require a major change to the design. It is not known at this time how the border wrapping Margolus functions could be modified to work on hardware without requiring excess connections to create the torus-like wrapping currently being used. Moreover, as Terry Ritter's research [31] was verified here, any random permutation of an 8-bit S-box will on average have a minimum non-linearity of 100. This means that there may be no point to using a CA to modify the S-box at all if you can programmatically generate random permutations (based on some key) more efficiently. This could remove the need for a generator table, the entire automata, the performance issues, and the ASIC deficiencies.

Having successfully solved some of the perceived issues with Rijndael's S-box and showing a secure use of CA in cryptography, the design is a success in all but performance and efficient hardware implementation. Showing one possible secure use of CA in cryptography was a major focus of this thesis, as was enhancing Rijndael's already powerful S-box. The ideas employed here, may be of interest to future cryptographers and cellular automata addicts alike.

5. REFERENCES

[1]   C. Zhang, Q. Peng, and Y. Li, "Encryption Based on Reversible Cellular Automata, IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions,", 2 ed 2002, pp. 1223-1226.

[2]   Jarkko Kari, "Cryptosystems Based on Reversible Cellular Automata, University of Turku, Finland," 1992.

[3]   Public Domain Images, "Cellular Automata," in *http://en.wikipedia.org/wiki/Cellular_automata* 2007.

[4]   Wolfram Mathworld, "Moore Neighborhood," in *http://mathworld.wolfram.com/MooreNeighborhood.html* 2007.

[5]   Wolfram Mathworld, "von Neumann Neighborhood," in *http://mathworld.wolfram.com/vonNeumannNeighborhood.html* 2003.

[6]   Jorg R.Weimer, "Reversible Cellular Automata," in *http://www.jweimar.de/jcasim/reversibleCA/* 2007.

[7]   Tim Tyler, "Margolus Neighborhood," in *http://www.cell-auto.com/neighbourhood/margolus/index.html* 2007.

[8]   Algred J.Menezes, Paul C.van Oorschot, and Scott A.Vanstone, "Introduction to Cryptography - Symmetric Key Encryption, Handbook of Applied Cryptography 5th Edition," 1996.

[9]   Joan Daemen and Vincent Rijmen, "AES Proposal - Rijndael, National Institute for Standards and Technology Consortium," 1999.

[10] Public Domain Images, "Wikipedia - Fiestel Cipher," in *http://en.wikipedia.org/wiki/Image:Feistel.png* 2007.

[11] Public Domain Images, "Wikipedia - AES," in *http://en.wikipedia.org/wiki/Advanced_Encryption_Standard* 2007.

[12] Public Domain Images, "Wikipedia - Block Cipher Modes of Operation," in *http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation* 2007.

[13] Algred J.Menezes, Paul C.van Oorschot, and Scott A.Vanstone, "Block Ciphers - Modes of Operation, Handbook of Applied Cryptography 5th Edition," 1996.

[14] T. Jamil, "The Rijndael algorithm," *Potentials, IEEE*, vol. 23, no. 2, pp. 36-38, 2004.

[15] C. Sanchez-Avila and R. Sanchez-Reillol, "The Rijndael block cipher (AES proposal) : a comparison with DES, 2001 IEEE 35th International Carnahan Conference on Security Technology," 2001, pp. 229-234.

[16] S. Nandi, B. K. Kar, and P. Pal Chaudhuri, "Theory and applications of cellular automata in cryptography," *Computers, IEEE Transactions on*, vol. 43, no. 12, pp. 1346-1357, 1994.

[17] S.Wolfram, "Cryptography with Cellular Automata," in *http://www.stephenwolfram.com/publications/articles/ca/85-cryptography/1/text.html* 1986.

[18] S.Wolfram, "Random Sequence Generation by Cellular Automata," in *http://www.stephenwolfram.com/publications/articles/ca/86-random/index.html* 1986.

[19] Sam Trenholme, "The AES Encryption Algorithm," in *http://www.samiam.org/rijndael.html* 2005.

[20] L. Jingmei, W. Baodian, C. Xiangguo, and W. Xinmei, "An AES S-box to Increase Complexity and Cryptographic Analysis, AINA 2005 - 19th International Conference on Advanced Information Networking and Applications,"*,* 1 ed 2005, pp. 724-728.

[21] Joanne Fuller and William Millan, "On Linear Redundancy in the AES S-Box, Queensland University of Technology - Information Security Research Centre," 2002.

[22] Tim Kerrins and Klaus Kursawe, "A Cautionary Note on Weak Implementations of Block Ciphers, Philips Research Europe - Information and System Security Group," 2007.

[23] Tommaso Toffoli and Norman Margolus, "Invertible Cellular Automata: A Review, Physica D,"*,* 45 ed 1990, pp. 229-253.

[24] B. Feng, "Cryptanalysis of a partially known cellular automata cryptosystem," *Computers, IEEE Transactions on*, vol. 53, no. 11, pp. 1493-1497, 2004.

[25] S.Wolfram, "A New Kind of Science," in *http://www.wolframscience.com/nksonline/* Wolfram Media, 2002, pp. 53-56.

[26] S. R. Blackburn, S. Murphy, K. G. Paterson, S. Nandi, and P. P. Chaudhuri, "Comments on "Theory and applications of cellular automata in cryptography" [and reply]," *Computers, IEEE Transactions on*, vol. 46, no. 5, pp. 637-639, 1997.

[27] S. Nandi and P. P. Chaudhuri, "Reply To Comments On "theory And Application Of Cellular Automata In Cryptography"," *Computers, IEEE Transactions on*, vol. 46, no. 5, p. 639, 1997.

[28] Debdeep Mukhopadhyay, "CASBox: A Programmable Structure to Generate S-Boxes Using Cellular Automata, Indian Institute of Technology," 2007.

[29] Isil Vergili and Melek D.Yucel, "Avalanche and Bit Independence Properties for the Ensembles of Randomly Chosen nxn S-Boxes, Turk J Elec Engin, Vol. 9, No. 2," 2001.

[30] Terry Ritter, "Measured Boolean Function Nonlinearity in Feistel Cipher Constructions," in *http://www.ciphersbyritter.com/ARTS/FEISNONL.HTM* 1998.

[31] Terry Ritter, "Measuring Boolean Function Nonlinearity by Walsh Transform," in *http://www.ciphersbyritter.com/ARTS/MEASNONL.HTM* 1998.

[32] Algred J.Menezes, Paul C.van Oorschot, and Scott A.Vanstone, "Maurer's Universal Statistical Test - Pseudorandom Bits and Sequences, Handbook of Applied Cryptography 5th Edition,"*, Chapter 5 ed 1996.

[33] Serge Vaudenay, "A Classical Introduction to Cryptography," in *Applications for Communication Security* Springer, 2006, pp. 17-18.

[34] Public Domain Images, "Information Entropy," in *http://en.wikipedia.org/wiki/Information_entropy* 2007.

[35] Public Domain Images, "Conditional Entropy," in *http://en.wikipedia.org/wiki/Conditional_entropy* 2007.

6. APPENDICES

APPENDIX A – Design Diagrams

A1. Galois Class (Rinjdael) Design

galois

- ltable : uint8_t
- etable : uint8_t
- gmulInv : uint8_t
- SBox : uint8_t
- SBoxInv : uint8_t
- key : uint8_t
- state : uint8_t
- IV : uint8_t
- IVTemp : uint8_t
- rounds : int
- encryptBool : bool
- runGensBeforeKey : bool
- galoisGenSeed : int
- generationsPerBlock : int
- keyGenerationsPerBlock : int
- threshCA : int
- ciphFile : char
- textFile : char
- userKey : char
- theCA : margolis*
- theKeyCA : margolisSingle*
- theTest : tests*

- + galois()
- + ~ galois()
- - gmul(a : uint8_t, b : uint8_t) : uint8_t
- - gmulLookup(a : uint8_t, b : uint8_t) : uint8_t
- - gmulInverse(in : uint8_t) : uint8_t
- - sub(in : uint8_t) : uint8_t
- - rcon(in : uint8_t) : uint8_t
- - ROTL(x : uint8_t, s : uint8_t) : uint8_t
- - ROTR(x : uint8_t, s : uint8_t) : uint8_t
- - gmixColumn(r : uint8_t*)
- - invGmixColumn(r : uint8_t*)
- - rotateStateL(offset : int)
- - rotateStateR(offset : int)
- - shiftRows()
- - invShiftRows()
- - mixColumns()
- - invMixColumns()
- - substituteBytes()
- - invSubstituteBytes()
- - addRoundKey(i : int)
- - textUserPrompt()
- - initCipher()
- - initIV()
- - doEncryptDecrypt()
- - generateLogTables()
- - generateGmulInverse()
- - generateSBoxes()
- - schedule_core(in : uint8_t*, i : uint8_t)
- - rotate(in : uint8_t*)
- - expand_key()
- - doRounds()
- - doInvRounds()
- - encrypt()
- - decrypt()
- - doCBCXOR()
- - doCBCUpdate(source[] : uint8_t, dest[] : uint8_t)
- - padState(padVal : int)
- - padFullState(padVal : int)
- - hexToInt(first : char, second : char) : uint8_t
- - verifyGenerators() : bool

margolis

0..1   -theCA

margolisSingle

0..1   -theKeyCA

tests

0..1   -theTest

78

A.2 Margolis Class (Automata)

margolisSingle
+ swap(x1 : int, y1 : int, x2 : int, y2 : int) + setParams(keyBoxz : uint8_t*, widthz : int, heightz : int, threshz : int)

margolis
SBox : uint8_t* # invSBox : uint8_t* # rules : bool # evenodd : bool # thresh : int # height : int # width : int
+ margolis() + ~ margolis() + setParams(sboxz : uint8_t*, invsboxz : uint8_t*, widthz : int, heightz : int, threshz : int) + doGenerations(numGens : int) # swap(x1 : int, y1 : int, x2 : int, y2 : int) # doTransition(x : int, y : int, currState : uint8_t) # getConfiguration(i : int, j : int) : uint8_t # setupRules()

A.3 Tests Class

tests
- sbox : uint8_t* - invsbox : uint8_t* - bits : int
+ tests(sboxz : uint8_t*, invsboxz : uint8_t*) + ~ tests() + testInvertibility() + testNonLinearity(verbose : bool) + testBitChanges(verbose : bool) + BitARFWT() + testDataHistogram(verbose : bool) + testMidPoints(verbose : bool) + testDistanceToCenter(verbose : bool) + testAvalanche(verbose : bool) + testAll(verbose : bool) + testEntropy(verbose : bool) + printSbox()

A.4 Complete Class Diagram

galois

- ltable : uint8_t
- etable : uint8_t
- gmulInv : uint8_t
- SBox : uint8_t
- SBoxInv : uint8_t
- key : uint8_t
- state : uint8_t
- IV : uint8_t
- IVTemp : uint8_t
- rounds : int
- encryptBool : bool
- runGensBeforeKey : bool
- galoisGenSeed : int
- generationsPerBlock : int
- keyGenerationsPerBlock : int
- threshCA : int
- ciphFile : char
- textFile : char
- userKey : char
- theCA : margolis*
- theKeyCA : margolisSingle*
- theTest : tests*

- + galois()
- + ~ galois()
- - gmul(a : uint8_t, b : uint8_t) : uint8_t
- - gmulLookup(a : uint8_t, b : uint8_t) : uint8_t
- - gmulInverse(in : uint8_t) : uint8_t
- - sub(in : uint8_t) : uint8_t
- - rcon(in : uint8_t) : uint8_t
- - ROTL(x : uint8_t, s : uint8_t) : uint8_t
- - ROTR(x : uint8_t, s : uint8_t) : uint8_t
- - gmixColumn(r : uint8_t*)
- - invGmixColumn(r : uint8_t*)
- - rotateStateL(offset : int)
- - rotateStateR(offset : int)
- - shiftRows()
- - invShiftRows()
- - mixColumns()
- - invMixColumns()
- - substituteBytes()
- - invSubstituteBytes()
- - addRoundKey(i : int)
- - textUserPrompt()
- - initCipher()
- - initIV()
- - doEncryptDecrypt()
- - generateLogTables()
- - generateGmulInverse()
- - generateSBoxes()
- - schedule_core(in : uint8_t*, i : uint8_t)
- - rotate(in : uint8_t*)
- - expand_key()
- - doRounds()
- - doInvRounds()
- - encrypt()
- - decrypt()
- - doCBCXOR()
- - doCBCUpdate(source[] : uint8_t, dest[] : uint8_t)
- - padState(padVal : int)
- - padFullState(padVal : int)
- - hexToInt(first : char, second : char) : uint8_t
- - verifyGenerators() : bool

margolis

- # SBox : uint8_t*
- # invSBox : uint8_t*
- # rules : bool
- # evenodd : bool
- # thresh : int
- # height : int
- # width : int

- + margolis()
- + ~ margolis()
- + setParams(sboxz : uint8_t*, invsboxz : uint8_t*, widthz : int, heightz : int, threshz : int)
- + doGenerations(numGens : int)
- # swap(x1 : int, y1 : int, x2 : int, y2 : int)
- # doTransition(x : int, y : int, currState : uint8_t)
- # getConfiguration(i : int, j : int) : uint8_t
- # setupRules()

margolisSingle

- + swap(x1 : int, y1 : int, x2 : int, y2 : int)
- + setParams(keyBoxz : uint8_t*, widthz : int, heightz : int, threshz : int)

-theKeyCA
0..1

tests

- - sbox : uint8_t*
- - invsbox : uint8_t*
- - bits : int

- + tests(sboxz : uint8_t*, invsboxz : uint8_t*)
- + ~ tests()
- + testInvertibility()
- + testNonLinearity(verbose : bool)
- + testBitChanges(verbose : bool)
- + BitARFWT()
- + testDataHistogram(verbose : bool)
- + testMidPoints(verbose : bool)
- + testDistanceToCenter(verbose : bool)
- + testAvalanche(verbose : bool)
- + testAll(verbose : bool)
- + testEntropy(verbose : bool)
- + printSbox()

.1-theTest

A.5 Function Source

Class	Function	Source			
		Original	Internet	Modified	
Galois	galois	X			
	~galois	X			
	gmul		X		
	gmulLookup			X	
	gmulInverse		X		
	rcon			X	
	ROTL	X			
	ROTR	X			
	gmixColumn			X	
	invGmixColumn			X	
	rotateStateL	X			
	rotateStateR	X			
	shiftRows	X			
	invShiftRows	X			
	mixColumns	X			
	invMixColumns	X			
	substituteBytes	X			
	invSubstituteBytes	X			
	addRoundKey	X			
	textUserPrompt	X			
	initCipher	X			
	initIV	X			
	doEncryptDecrypt	X			
	generateLogTables			X	
	generateGmulInverse		X		
	generateSBoxes		X		
	schedule_core		X		
	rotate	X			
	expand_key			X	
	doRounds	X			
	doInvRounds	X			
	encrypt	X			
	decrypt	X			
	doCBCXOR	X			
	doCBCUpdate	X			
	padState	X			
	padFullState	X			
	hexToInt			X	
	verifyGenerators	X			
Margolis	margolis	X			
	~margolis	X			
	setParams	X			
	doGenerations	X			
	swap	X			
	doTransition	X			
	getConfiguration	X			
	setupRules	X			
Tests	tests	X			
	~tests	X			
	testInvertibility			X	
	testNonLinearity			X	
	testBitChanges	X			
	BitARFWT		X		
	testDataHistogram	X			
	testMidPoints	X			
	testDistanceToCenter	X			
	testAvalanche	X			
	testAll	X			
	testEntropy	X			
	printSbox	X			
	Totals	45	6	9	**60**
	Percentage	75	10.0	15.0	**100**

A.6 Functional Flowchart
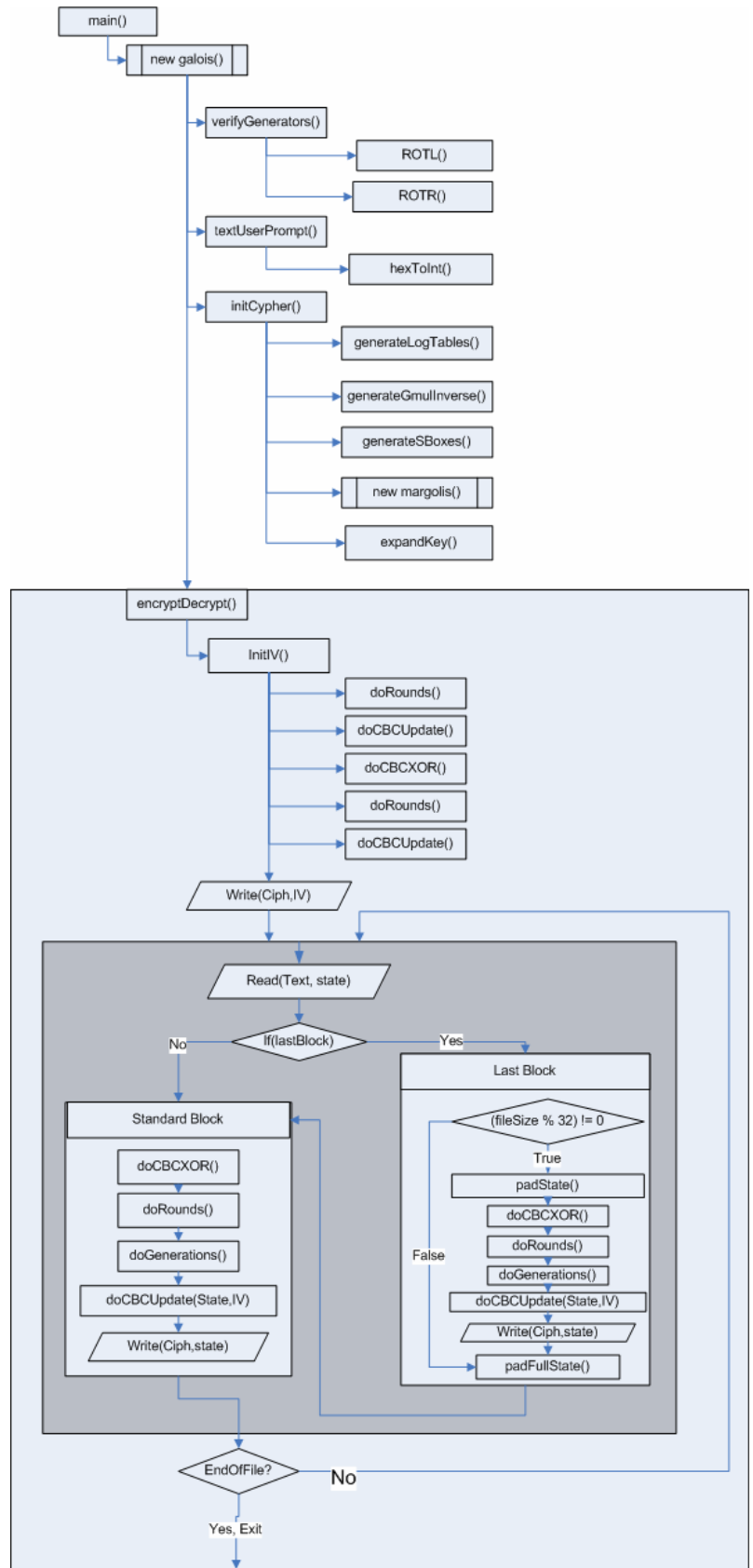
APPENDIX B – Implementation Source

B1. Galois Class Header

```
#ifndef GALOIS_H
#define GALOIS_H

#include <iostream>
#include <fstream>
#include <cassert>
#include <sys/stat.h>
#include <time.h>

#include "margolis.h"
#include "tests.h"

using std::cout;
using std::endl;
using std::ios;
using std::hex;
using std::dec;
using std::ifstream;
using std::ofstream;
using std::cin;
using std::fill;

const uint8_t generators[] =
    { /* The Galois Generator Table */
      0x03, 0x05, 0x06, 0x09, 0x0b, 0x0e, 0x11, 0x12, 0x13, 0x14, 0x17,
0x18, 0x19, 0x1a, 0x1c, 0x1e, 0x1f, 0x21, 0x22, 0x23, 0x27, 0x28, 0x2a,
0x2c, 0x30, 0x31, 0x3c, 0x3e, 0x3f, 0x41, 0x45, 0x46, 0x47, 0x48, 0x49,
0x4b, 0x4c, 0x4e, 0x4f, 0x52, 0x54, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b,
0x5f, 0x64, 0x65, 0x68, 0x69, 0x6d, 0x6e, 0x70, 0x71, 0x76, 0x77, 0x79,
0x7a, 0x7b, 0x7e, 0x81, 0x84, 0x86, 0x87, 0x88, 0x8a, 0x8e, 0x8f, 0x90,
0x93, 0x95, 0x96, 0x98, 0x99, 0x9b, 0x9d, 0xa0, 0xa4, 0xa5, 0xa6, 0xa7,
0xa9, 0xaa, 0xac, 0xad, 0xb2, 0xb4, 0xb7, 0xb8, 0xb9, 0xba, 0xbe, 0xbf,
0xc0, 0xc1, 0xc4, 0xc8, 0xc9, 0xce, 0xcf, 0xd0, 0xd6, 0xd7, 0xda, 0xdc,
0xdd, 0xde, 0xe2, 0xe3, 0xe5, 0xe6, 0xe7, 0xe9, 0xea, 0xeb, 0xee, 0xf0,
0xf1, 0xf4, 0xf5, 0xf6, 0xf8, 0xfb, 0xfd, 0xfe, 0xff };
```

```cpp
class galois
{
public:
    galois(); //Default constructor
    ~galois(); //Default Destructor

private:
    uint8_t gmul ( uint8_t a, uint8_t b ); //Function for multiplying
two numbers without lookup table
    uint8_t gmulLookup ( uint8_t a, uint8_t b ); //Function for
multiplying two numbers with lookup table
    uint8_t gmulInverse ( uint8_t in ); //Function for calculating the
multiplicative inverse of a number
    uint8_t sub ( uint8_t in ); //Function for substituting a byte with
the sbox
    uint8_t rcon ( uint8_t in ); //Function for exponentiating 2^in
used in key expansion
    uint8_t ROTL ( uint8_t x, uint8_t s ); //Function for left circular
shifting an 8bit number s times
    uint8_t ROTR ( uint8_t x, uint8_t s ); //Function for right
circular shifting an 8bit number s times

    void gmixColumn ( uint8_t *r ); //Rijndaels MixColumn step
    void invGmixColumn ( uint8_t *r ); //Rijndaels Inverse MixColumn
step
    void rotateStateL ( int offset ); //RotateStateLeft, takes offset
into one dimensional state variable
    void rotateStateR ( int offset ); //RotateStateRight, takes offset
into one dimensional state variable
    void shiftRows(); //Calls RotateStateL to perform the neccessary
shifts for 256-bit blocks
    void invShiftRows(); //Calls RotateStateR to perform the neccessary
unshifts for 256-bit blocks
    void mixColumns(); //Calls gMixColumn to perform mixing on each
column
    void invMixColumns(); //Calls invGmixColumn to perform unmixing on
each column
    void substituteBytes(); //Substitutes bytes with their SBox lookup
    void invSubstituteBytes(); //UnSubstitutes bytes with their invSBox
lookup
    void addRoundKey ( int i ); //Adds (xors) the indicated roundkey to
the state

    void textUserPrompt(); //Prompts the user for required input
    void initCipher(); //Initializes the core functionality of the
cipher
    void initIV(); //Initializes the IV for CBC Mode
    void doEncryptDecrypt(); //Calls encrypt or decrypt function based
on user input
    void generateLogTables(); //Generates the Logarithm tables for the
Galois Field
    void generateGmulInverse(); //Generates the gmulInverse table
    void generateSBoxes(); //Generates the SBox and InvSBox
    void schedule_core ( uint8_t *in, uint8_t i ); //Function used in
key expansion
```

```
    void rotate ( uint8_t *in ); //Rotate function used in key
expansion
    void expand_key(); //Key Expansion function
    void doRounds(); //Rijndaels doRounds function
    void doInvRounds(); //Rijndaels doInvRounds function
    void encrypt(); //Encryption function with File IO and Padding
    void decrypt(); //Decryption function with File IO and DePadding
    void doCBCXOR(); //CBC function to XOR over the state
    void doCBCUpdate ( uint8_t source[32], uint8_t dest[32] ); //CBC
function for updating the IV
    void padState ( int padVal ); //Function for Padding the State
    void padFullState ( int padVal ); //Function for adding an entire
pad block

    uint8_t hexToInt ( char first, char second ); //Function for
converting user hex key input into real hex numbers
    bool verifyGenerators(); //Function for verifying the static
generator table

    uint8_t ltable[256]; //Log Table
    uint8_t etable[256]; //Exponentiation Table
    uint8_t gmulInv[256]; //Multiplicative Inverse Table
    uint8_t SBox[256]; //SBox
    uint8_t SBoxInv[256]; //Inverse SBox
    uint8_t key[480]; //The Key and expanded key space

    uint8_t state[32]; //The state, or current block
    uint8_t IV[32]; //The initialization vector
    uint8_t IVTemp[32]; //Space for a temporary IV
    int rounds; //Number of rounds in the encryption

    bool encryptBool; //Boolean from user input
    bool runGensBeforeKey; //CA Key expansion modifying boolean from
user input
    int galoisGenSeed; //Galois Generator seed from user input
    int generationsPerBlock; //CA Generations from user input
    int keyGenerationsPerBlock; //KeyCA Generations from user input
    int threshCA; //CA Threshold from user input
    char ciphFile[33]; //Filename for ciphertext
    char textFile[33]; //Filename for plaintext
    char userKey[32]; //User input key space

    margolis *theCA; //The SBox and InvSBox Modifying CA
    margolisSingle *theKeyCA; //The Key modifying CA
    tests *theTest; //Object for the tests class
};

#include "galois.cpp"

#endif
```

B2. Galois Class Source File

```
#ifndef GALOIS_CPP
#define GALOIS_CPP
```

```cpp
#include "galois.h"

galois::galois()
{
    /*Default Constructor*/
    verifyGenerators(); //Verify the Generator Tables

    /*Blank all arrays*/
    fill ( key, key + 480, 0 );
    fill ( ltable, ltable + 256, 0 );
    fill ( etable, etable + 256, 0 );
    fill ( gmulInv, gmulInv + 256, 0 );
    fill ( SBox, SBox + 256, 0 );
    fill ( SBoxInv, SBoxInv + 256, 0 );
    fill ( state, state + 32, 0 );
    fill ( IV, IV + 32, 0 );
    fill ( IVTemp, IVTemp + 32, 0 );
    fill ( userKey, userKey + 32, 0 );
    fill ( ciphFile, ciphFile + 33, 0 );
    fill ( textFile, textFile + 33, 0 );

    textUserPrompt(); //Prompt for User Parameters

    initCipher();  //Initialize the cipher
    doEncryptDecrypt(); //Perform User Function
}

void galois::initCipher()
{
    rounds = 14;  //Set Default Rounds
    generateLogTables(); //Generate the Log Tables
    generateGmulInverse(); //Generate the Multiplicative Inverse Tables
    generateSBoxes();  //Generate the S-Box and Inverse S-Box
    theCA = new margolis(); //Create the S-Box CA instance
    theCA->setParams ( SBox, SBoxInv, 16, 16, threshCA ); //Set the
user chosen parameters

    theTest = new tests ( SBox, SBoxInv ); //Create the Tests class

    if ( runGensBeforeKey ) //If user chose to modify key expansion
    {
        theCA->doGenerations ( generationsPerBlock ); //Modify the S-
Box during initialization
    }

    expand_key(); //Expand the Users Key

    theKeyCA = new margolisSingle(); //Create the Key CA instance
    theKeyCA->setParams ( key, 24, 20, threshCA ); //Set the user
chosen parameters
}

uint8_t galois::hexToInt ( char first, char second )
{
    /*Method for converting ascii hex characters to numbers*/
    char hex[3];
```

```cpp
    char *stop;
    hex[0] = first;
    hex[1] = second;
    hex[2] = 0;
    return strtol ( hex, &stop, 16 );
}

void galois::textUserPrompt()
{
    /*Prompt the User for their options*/
    char input[129];
    fill ( input, input + 129, 0 );

    char choice = 0;

    do
    {
        cout << "Encrypt or Decrypt? (e,d) " << endl;
        choice = cin.get();
    }

    while ( ( choice != 'e' ) && ( choice != 'd' ) );

    cin.ignore();

    encryptBool = ( choice == 'e' );

    do
    {
        cout << "Would you like to enter a 1)Passphrase or 2)Key? " <<
endl;
        choice = cin.get();
    }

    while ( ( choice != '1' ) && ( choice != '2' ) );

    cin.ignore();

    if ( choice == '1' )
    {
        cout << "***Input 32 Ascii Characters****" << endl;
        cin.getline ( input, 33 );

        for ( int i = 0;i < 32;i++ )
        {
            userKey[i] = input[i];
        }
    }

    else
    {
        do
        {
            cout << "***********************Input 64 Hex Digit
Key******************" << endl;
            cin.getline ( input, 65 );
        }
```

```cpp
        while ( cin.gcount() != 65 );

        for ( int i = 0;i < 64;i += 2 )
        {
            userKey[i/2] = hexToInt ( input[i], input[i+1] );
        }
    }

    do
    {
        cout << "***TextFile Name: <=32 Chars****" << endl;
        cin.getline ( textFile, 33 );
    }

    while ( cin.gcount() == 1 );

    do
    {
        cout << "***CiphFile Name: <=32 Chars****" << endl;
        cin.getline ( ciphFile, 33 );
    }

    while ( cin.gcount() == 1 );

    do
    {
        cout << "Enter your galois generator number (0-127)" << endl;
        cin >> galoisGenSeed;
    }

    while ( ( galoisGenSeed > 127 ) || ( galoisGenSeed < 0 ) );

    cin.ignore();

    galoisGenSeed = generators[galoisGenSeed];

    do
    {
        cout << "Enter the number of generations per block: " << endl;
        cin >> generationsPerBlock;
        cin.ignore();
    }

    while ( ( generationsPerBlock < 0 ) || ( generationsPerBlock > 128
) );

    do
    {
        cout << "Enter the number of key generations per block: " <<
endl;
        cin >> keyGenerationsPerBlock;
        cin.ignore();
    }

    while ( ( keyGenerationsPerBlock < 0 ) || ( keyGenerationsPerBlock
> 128 ) );
```

```
    if ( generationsPerBlock > 0 ) do
        {
            cout << "Enter the threshold value for the Automata: (64-
192) (127 RECOMMENDED) " << endl;
            cin >> threshCA;
            cin.ignore();
        }

        while ( ( threshCA <= 63 ) || ( threshCA >= 193 ) );

    choice = 'n';

    if ( generationsPerBlock > 0 ) do
        {
            cout << "Would you like to run (" << dec <<
generationsPerBlock << ") generations before key expansion? (y,n) " <<
endl;
            choice = cin.get();
            cin.ignore();
        }

        while ( ( choice != 'y' ) && ( choice != 'n' ) );

    runGensBeforeKey = ( choice == 'y' );
}

galois::~galois()
{
    /*Default Destructor, Delete instantiated objects*/
    delete theCA;
    delete theKeyCA;
    delete theTest;
}

void galois::initIV()
{
    /*CBC Initialization Vector initilization Method*/
    clock_t start_tick; //Create clock object

    for ( int x = 0;x < 32;x++ ) //For the size of our block
    {
        state[x] = 0; //Blank Entry
        start_tick = clock(); //Set clocks current time

        while ( clock() == start_tick ) //While still current time
        {
            state[x]++; //Increment this entry in the block
        }
    }

    doRounds(); //Encrypt the state with the users key
    doCBCUpdate ( state, IV ); //Update the IV

    for ( int x = 0;x < 32;x++ ) //For the size of our block
    {
        state[x] = 0; //Blank Entry
```

```cpp
        start_tick = clock(); //Set clocks current time

        while ( clock() == start_tick ) //While still current time
        {
            state[x]++; //Increment this entry in the block
        }
    }

    doCBCXOR(); //CBC XOR with our first IV
    doRounds(); //Encrypt the IV with users key

    doCBCUpdate ( state, IV ); //Update the FINAL IV
}

void galois::doEncryptDecrypt()
{
    /*Method for simply calling Encrypt or Decrypt based on user
input*/
    if ( encryptBool ) encrypt();
    else decrypt();
}

void galois::encrypt()
{
    /*The Encryption Method*/
    initIV(); //Initialize the IV

    struct stat results; //Get the FileSize
    int filesize = stat ( textFile, &results );
    filesize = results.st_size;

    ifstream text ( textFile, ios::in | ios::binary ); //Open the Input
and Output Files
    ofstream ciph ( ciphFile, ios::out | ios::binary );

    assert ( text ); //Assert that the files were opened
    assert ( ciph );

    if ( filesize == 0 ) //Refused to Encrypt an Empty File
    {
        cout << "Input file size is zero! Not Encrypting " << endl;
        exit ( 0 );
    }

    ciph.write ( ( char* ) IV, sizeof ( IV ) ); //Write the IV as the
first block of the ciphertext

    for ( int z = 0;z <= filesize;z += 32 ) //For all blocks in the
file
    {
        text.read ( ( char* ) state, sizeof ( state ) ); //Read a block
of text

        if ( z + 32 > filesize ) //If last block
        {
            int padVal = 32 - ( filesize % 32 ); //Calculate the number
of bytes needed to fill the last block
```

```
                if ( padVal != 32 ) //If the file didn't end as a perfect
multiple of the block size (32)
                {
                    padState ( padVal ); //Pad the Last Block to fill it
out
                    doCBCXOR(); //Do CBC XOR as usual
                    doRounds(); //Do Encryption rounds as Usual
                    theCA->doGenerations ( generationsPerBlock ); //Run the
CA on the SBox as usual
                    theKeyCA->doGenerations ( keyGenerationsPerBlock );
//Run the CA on the Key as usual
                    doCBCUpdate ( state, IV ); //Update the IV with
encrypted block
                    ciph.write ( ( char* ) state, sizeof ( state ) );
//Output the last block
                    padFullState ( padVal ); //Prepare the final padding
block
                }
                else
                {
                    padFullState ( 0 ); //Only adding the final padding
block because file was a multiple of block size
                }
            }

        doCBCXOR(); //Either doing normal processing or finishing final
padding block
        doRounds(); //Do Encryption rounds
        theCA->doGenerations ( generationsPerBlock ); //Do S-Box
Generations
        theKeyCA->doGenerations ( keyGenerationsPerBlock ); //Do Key
Generations
        doCBCUpdate ( state, IV ); //Update the IV with the encrypted
block

        ciph.write ( ( char* ) state, sizeof ( state ) ); //Output the
block
    }

    text.close(); //Close the files
    ciph.close();
}

void galois::padState ( int padVal )
{
    /*Method to pad a block to fill it up with padVal + 32*/
    for ( int i = 31;i >= 32 - padVal;i-- ) //Starting from the end,
write as many as needed
    {
        state[i] = padVal + 32; //Fill with 32 + Count
    }
}

void galois::padFullState ( int padVal )
{
    /*Method to pad a full block with 32 + Count*/
```

```
    for ( int i = 31;i >= 0;i-- )
    {
        state[i] = padVal + 32; //PadVal will be zero if the file was a
multiple of 32
    }
}

void galois::decrypt()
{
    /*Method for encrypting a file*/

    struct stat results; //Get the FileSize
    int filesize = stat ( ciphFile, &results );
    filesize = results.st_size - 64; //Subtract the IV and Final
Padding block from the calculations

    ifstream ciph ( ciphFile, ios::in | ios::binary ); //Open the files
    ofstream text ( textFile, ios::out | ios::binary );

    assert ( ciph ); //Verify the files are open
    assert ( text );

    ciph.read ( ( char* ) IV, sizeof ( IV ) ); //Read the IV from the
first block of the ciphertext

    for ( int z = 0;z < filesize;z += 32 ) //For all blocks in the file
    {
        ciph.read ( ( char* ) state, sizeof ( state ) ); //Read in an
ciphertext block

        doCBCUpdate ( state, IVTemp ); //Backup this block as the next
IV
        doInvRounds(); //Do the Inverse Rounds
        theCA->doGenerations ( generationsPerBlock ); //Do S-Box CA
Generations
        theKeyCA->doGenerations ( keyGenerationsPerBlock ); //Do Key CA
Generations
        doCBCXOR(); //Do CBC XOR
        doCBCUpdate ( IVTemp, IV ); //Update the IV with the temp IV

        if ( z + 32 >= filesize ) //If Final Block
        {
            uint8_t finalState[32]; //Create a holding space for the
final state
            memcpy ( finalState, state, 32 ); //Copy the state into
finalState

            ciph.read ( ( char* ) state, sizeof ( state ) ); //Read the
Final Padding Block
            doCBCUpdate ( state, IVTemp ); //Update the IVTemp with the
state
            doInvRounds(); //Do the Inverse Rounds
            theCA->doGenerations ( generationsPerBlock ); //Do S-Box CA
Generations
            theKeyCA->doGenerations ( keyGenerationsPerBlock ); //Do
Key CA Generations
            doCBCXOR(); //Do CBC XOR
```

```cpp
            if ( state[0] == 32 ) //If the final padding block's last
entry is 32
            {
                text.write ( ( char* ) finalState, 32 ); //Write the
ENTIRE final state out, original filesize % 32 was 0
            }
            else
            {
                text.write ( ( char* ) finalState, 32 - ( state[0] % 32
) ); //Otherwise write out the number of bytes that weren't padding
            }
        }
        else
        {
            text.write ( ( char* ) state, sizeof ( state ) ); //Write
out a normal block of decrypted plain-text, wasnt last block.
        }
    }

    text.close(); //Close files
    ciph.close();
}

void galois::doCBCXOR()
{
    /*Do the XOR operation of CBC Mode*/
    for ( int i = 0;i < 32;i++ ) state[i] ^= IV[i]; //XOR each state
entry with the IV
}

void galois::doCBCUpdate ( uint8_t source[32], uint8_t dest[32] )
{
    /*Wrapper Method for copying to and from state, IV, IVTemp*/
    memcpy ( dest, source, 32 );
}

void galois::doRounds()
{
    /*Rijndael DoRounds*/
    addRoundKey ( 0 ); //Add Key 0

    for ( int i = 1;i < rounds;i++ ) //For all rounds
    {
        substituteBytes(); //S-Box
        shiftRows(); //Shift
        mixColumns(); //Mix
        addRoundKey ( i ); //Add Key i
    }

    substituteBytes(); //Final Sbu
    shiftRows(); //Final Shift
    addRoundKey ( 14 ); //Final Key
}

void galois::doInvRounds()
{
```

```
    addRoundKey ( 14 ); //Add Key 14
    invShiftRows(); //UnShift

    for ( int i = rounds - 1;i > 0;i-- ) //For all Rounds in reverse
    {
        invSubstituteBytes(); //Inv S-Box
        addRoundKey ( i ); //Add Key i
        invMixColumns(); //UnMix
        invShiftRows(); //UnShift
    }

    invSubstituteBytes(); //Inv S-Box

    addRoundKey ( 0 ); //Add Key 0
}

void galois::substituteBytes()
{
    /*Substitute all bytes in state with their S-Box entries*/
    for ( int i = 0;i < 32;i++ )
    {
        state[i] = SBox[state[i]];
    }
}

void galois::invSubstituteBytes()
{
    /*Substitute all bytes in the state with their Inverse S-Box
entries*/
    for ( int i = 0;i < 32;i++ )
    {
        state[i] = SBoxInv[state[i]];
    }
}

void galois::addRoundKey ( int round )
{
    /*XOR Key block with state block*/
    int location = 32 * round; //Calculate the offset in the expanded
key

    for ( int i = 0;i < 32;i++ ) //For all entries in the key block
    {
        state[i] ^= key[location++]; //Do XOR (addition)
    }
}

uint8_t galois::ROTL ( uint8_t x, uint8_t s )
{
    /*Circular rotate a byte left by s bits*/
    return ( uint8_t ) ( ( ( x ) << ( s& ( 8 - 1 ) ) ) | ( ( x ) >> ( 8
- ( s& ( 8 - 1 ) ) ) ) );
}

uint8_t galois::ROTR ( uint8_t x, uint8_t s )
{
    /*Circular rotate a byte right by s bits*/
```

```
    return ( uint8_t ) ( ( ( x ) >> ( s& ( 8 - 1 ) ) ) | ( ( x ) << ( 8
- ( s& ( 8 - 1 ) ) ) ) );
}

bool galois::verifyGenerators()
{
    /*Method to algorithmically verfiy the generator tables*/
    uint8_t check = 0xb7; //Default Check Value

    for ( int c = 0; c < 128; c++ ) //For all entries in the generator
table
    {
        check ^= ROTL ( generators[c], ROTR ( generators[127-c], c ) );
//Do some Mixing
    }

    uint8_t compare = 1; //Algorithmically build compare

    compare = ROTL ( compare, 2 );
    compare |= ROTL ( compare, 2 );
    compare |= ROTR ( compare, 6 );

    if ( check != compare ) //If not equal, tables modified
    {
        cout << "Error! Generator table has been altered! Check: " <<
hex << ( uint16_t ) check << " Compare: " << hex << ( uint16_t )
compare << endl;
    }

    else
    {
        cout << "Generator Tables Verified! Check: " << hex << (
uint16_t ) check << " Compare: " << hex << ( uint16_t ) compare <<
endl;
    }
}

uint8_t galois::gmul ( uint8_t a, uint8_t b )
{
    /*Perform multiplication in the galois field*/
    uint8_t p = 0;
    uint8_t hi_bit_set;

    for ( int counter = 0; counter < 8; counter++ )
    {
        if ( ( b & 1 ) == 1 )
            p ^= a;

        hi_bit_set = ( a & 0x80 );

        a <<= 1;

        if ( hi_bit_set == 0x80 )
            a ^= 0x1b;

        b >>= 1;
    }
```

```
    return p;
}

void galois::generateLogTables()
{
    /*Generate Log Tables with the given Generator Seed*/
    etable[0] = 1;
    etable[255] = 1;

    for ( int c = 1;c < 256;c++ )
    {
        etable[c] = gmul ( etable[c-1], galoisGenSeed );
        ltable[etable[c]] = c;
    }
}

/*Very fast define Method for performing multiplications with the
lookup tables*/
#define gmulLookupDefine(a, b)
((a==0)?0:etable[(ltable[a]+ltable[b])%255])

uint8_t galois::gmulLookup ( uint8_t a, uint8_t b )
{
    /*Slow Method for performing multiplication with the lookup
tables*/
    return ( a == 0 ) ? 0 : etable[ ( ltable[a] + ltable[b] ) % 255];
}

uint8_t galois::gmulInverse ( uint8_t in )
{
    /*Method for calculating the multiplicative inverse from the lookup
tables*/
    if ( in == 0 ) return 0;
    else return etable[ ( 255 - ltable[in] ) ];
}

void galois::generateGmulInverse()
{
    /*Method for generating the gmulInverse table*/
    gmulInv[0] = 0;

    for ( int c = 1;c < 256;c++ ) //For all 256 divisors
    {
        gmulInv[c] = gmulInverse ( c ); //Populate the gmulInv table
    }
}

uint8_t galois::sub ( uint8_t in )
{
    /*Method for generating the S-box entries based on the Generator
Seed*/
    uint8_t s, x;
    s = x = gmulInv[in];

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

```
            s = ( s << 1 ) | ( s >> 7 );
            x ^= s;
    }

    x ^= galoisGenSeed;

    return x;
}

void galois::generateSBoxes()
{
    /*Method for generating the S-Box tables using sub*/
    for ( int c = 0;c < 256;c++ ) //For each byte value 0 - 255
    {
        SBox[c] = sub ( c ); //Populate the S-Box with its sub value
        SBoxInv[SBox[c]] = c; //Populate the SBoxInverse location of
the sub value with the byte value
    }
}

void galois::mixColumns()
{
    /*Method for performing gmixColumn on all columns of the state*/
    uint8_t temp[4];

    for ( int j = 0;j < 8;j++ ) //For all 8 columns
    {
        for ( int i = 0;i < 4;i++ ) //For all 4 entries in a column
        {
            temp[i] = state[i*8+j]; //Copy them from the state to temp
        }

        gmixColumn ( temp ); //Mix temp

        for ( int i = 0;i < 4;i++ ) //For all 4 entries in a column
        {
            state[i*8+j] = temp[i]; //Restore them into the state from
temp
        }
    }
}

void galois::invMixColumns()
{
    /*Method for performing invGmixColumn on all columns of the state*/
    uint8_t temp[4];

    for ( int j = 0;j < 8;j++ ) //For all 8 columns
    {
        for ( int i = 0;i < 4;i++ ) //For all 4 entries in a column
        {
            temp[i] = state[i*8+j]; //Copy them from the state to temp
        }

        invGmixColumn ( temp ); //UnMix temp

        for ( int i = 0;i < 4;i++ ) //For all 4 entries in a column
```

```
        {
            state[i*8+j] = temp[i]; //Restore them into the state from
temp
        }
    }
}

void galois::gmixColumn ( uint8_t *r )
{
    /*Perform mixColumn on a single column*/

    uint8_t a[4]; //Make holding location
    memcpy ( a, r, 4 ); //Copy original into holding

//Perform matrix multiplication by {{2,3,1,1}[15;15;15]}
    r[0] = gmulLookupDefine ( a[0], 2 ) ^ gmulLookupDefine ( a[3], 1 )
^ gmulLookupDefine ( a[2], 1 ) ^ gmulLookupDefine ( a[1], 3 );
    r[1] = gmulLookupDefine ( a[1], 2 ) ^ gmulLookupDefine ( a[0], 1 )
^ gmulLookupDefine ( a[3], 1 ) ^ gmulLookupDefine ( a[2], 3 );
    r[2] = gmulLookupDefine ( a[2], 2 ) ^ gmulLookupDefine ( a[1], 1 )
^ gmulLookupDefine ( a[0], 1 ) ^ gmulLookupDefine ( a[3], 3 );
    r[3] = gmulLookupDefine ( a[3], 2 ) ^ gmulLookupDefine ( a[2], 1 )
^ gmulLookupDefine ( a[1], 1 ) ^ gmulLookupDefine ( a[0], 3 );
}

void galois::invGmixColumn ( uint8_t *r )
{
    /*Perform invGmixColumn on a single column*/

    uint8_t a[4]; //Make holding location
    memcpy ( a, r, 4 ); //Copy original into holding

//Perform matrix multiplcation by
{{14,9,13,11}{11,14,9,13}{13,11,14,9}{9,13,11,14}}
    r[0] = gmulLookupDefine ( a[0], 14 ) ^ gmulLookupDefine ( a[3], 9 )
^ gmulLookupDefine ( a[2], 13 ) ^ gmulLookupDefine ( a[1], 11 );
    r[1] = gmulLookupDefine ( a[1], 14 ) ^ gmulLookupDefine ( a[0], 9 )
^ gmulLookupDefine ( a[3], 13 ) ^ gmulLookupDefine ( a[2], 11 );
    r[2] = gmulLookupDefine ( a[2], 14 ) ^ gmulLookupDefine ( a[1], 9 )
^ gmulLookupDefine ( a[0], 13 ) ^ gmulLookupDefine ( a[3], 11 );
    r[3] = gmulLookupDefine ( a[3], 14 ) ^ gmulLookupDefine ( a[2], 9 )
^ gmulLookupDefine ( a[1], 13 ) ^ gmulLookupDefine ( a[0], 11 );
}

void galois::rotateStateR ( int offset )
{
    /*Method used by shiftRows to scramble rows*/

    uint8_t temp = state[offset+7]; //Backup last character

    for ( int c = 7;c > 0;c-- ) //Shift all once
    {
        state[offset + c] = state[offset + c - 1];
    }

    state[offset] = temp; //Restore first character (wrapped)
}
```

```cpp
void galois::rotateStateL ( int offset )
{
    /*Method used by invShiftRows to unscramble rows*/

    uint8_t temp = state[offset]; //backup first character

    for ( int c = 0;c < 7;c++ ) //Shift all once
    {
        state[offset + c] = state[offset + c + 1];
    }

    state[offset+7] = temp; //Restore last Character (wrapped)
}

void galois::rotate ( uint8_t *in )
{
    /*4-byte Rotate method for use in Key Expansion*/
    uint8_t temp = in[0];

    for ( int c = 0;c < 3;c++ )
    {
        in[c] = in[c + 1];
    }

    in[3] = temp;
}

void galois::shiftRows()
{
    /*ShiftRows method for performing Rijndaels shiftrow for 256-bit
blocks*/
    for ( int i = 1;i < 4;i++ ) //For all 3 rows to be shifted (0 is
never shifted)
    {
        switch ( i )
        {

        case 3:
            rotateStateL ( i*8 ); //Shift Row 3 4 times passing offset
into state

        case 2:
            rotateStateL ( i*8 );

            rotateStateL ( i*8 ); //Shift Row 2 3 times

        case 1:
            rotateStateL ( i*8 ); //Shift Row one once
        }
    }
}

void galois::invShiftRows()
{
    /*InvShiftRows method for performing Rijndaels unshiftrow for 256-
bit blocks*/
```

```
    for ( int i = 1;i < 4;i++ ) //For all 3 rows to be shifted (0 is
never shifted)
    {
        switch ( i )
        {

        case 3:
            rotateStateR ( i*8 ); //UnShift Row 3 4 times passing
offset into state

        case 2:
            rotateStateR ( i*8 );

            rotateStateR ( i*8 ); //UnShift Row 2 3 times

        case 1:
            rotateStateR ( i*8 ); //UnShift Row 1 once
        }
    }
}

uint8_t galois::rcon ( uint8_t in )
{
    /*Method performs the rcon operation (2 exponentiated) for key
expansion*/
    uint8_t c = 1;

    if ( in == 0 ) return 0; //Anything to the 0 is 0

    while ( in != 1 )
    {
        c = gmulLookupDefine ( c, 2 ); //Continuous multiply by 2
        in--;
    }

    return c;
}

void galois::schedule_core ( uint8_t *in, uint8_t i )
{
    /*Schedule core is used in key expansion*/

    rotate ( in ); //Rotate all 4 bytes

    for ( int a = 0; a < 4; a++ ) //Substitute all 4 bytes
    {
        in[a] = SBox[in[a]];
    }

    in[0] ^= rcon ( i ); //XOR with 2 exponentiated to some power
}

void galois::expand_key()
{
    /*Key Expansion Method*/
    for ( int i = 0;i < 32;i++ )
```

```
    {
         key[i] = userKey[i]; //Set first 32 bytes to be the user
inputted key
    }

    uint8_t t[4]; //Create temp 4 bytes

    int c = 32; //Start after the user inputted key
    uint8_t i = 1; //rcon exponentiation values stats at 1

    while ( c < 480 ) //For all expanded eky bytes
    {
         for ( int a = 0; a < 4; a++ ) //Base the first 4 bytes on the
previous key blocks first 4 bytes
         {
             t[a] = key[a + c - 4];
         }

         if ( c % 32 == 0 ) //If processing the end of a key block
         {
             schedule_core ( t, i ); //Call schedule core to modify the
4 bytes in t
             i++;
         }

         if ( c % 32 == 16 ) //If processing the middle of a key block
         {
             for ( int a = 0; a < 4; a++ ) //Do a Subsitute for all 4
bytes
             {
                 t[a] = SBox[t[a]];
             }
         }

         for ( int a = 0; a < 4; a++ ) //For all 4 bytes
         {
             key[c] = key[c - 32] ^ t[a]; //Key keys current entry is
the result of xoring the last blocks 4 bytes and t
             c++;
         }
    }
}

#endif
```

B3. Margolis Class Header File

```
#ifndef MARGOLIS_H
#define MARGOLIS_H

#define getCellInv(x,y) invSBox[SBox[fixX(x)+fixY(y)*width]] //Define
for getting value from InvSBox
#define getCell(x,y) SBox[fixX(x)+fixY(y)*width] //Define for getting
vlue from SBox
```

```
#define fixX(x) ((width + (x))%width) //Define for Fixing X values
(border wrapping)
#define fixY(y) ((height + (y))%height) //Define for Fixing Y values
(border wrapping)

#define swapDiag1(x,y) swap(x,y,x+1,y+1) //Defines for calling swap
Method for different swaps
#define swapDiag2(x,y) swap(x+1,y,x,y+1)
#define swapHoriz1(x,y) swap(x,y,x+1,y)
#define swapHoriz2(x,y) swap(x,y+1,x+1,y+1)
#define swapVert1(x,y) swap(x,y,x,y+1)
#define swapVert2(x,y) swap(x+1,y,x+1,y+1)

class margolis //Basic functionality Margolis Class for use with SBox
and InvSBox (see MargolisSingle)
{
public:
    margolis(); //Default Constructor
    ~margolis(); //Default Destructor
    virtual void setParams ( uint8_t *sboxz, uint8_t *invsboxz, int
widthz, int heightz, int threshz ); //Method for setting parameters for
normal Margolis
    void doGenerations ( int numGens ); //Method for running N number
of generations

protected:
    virtual void swap ( int x1, int y1, int x2, int y2 ); //Method for
performing swaps of values in neighborhood
    void doTransition ( int x, int y, uint8_t currState ); //Method for
performing the neccessary transition rules for the current
configuration
    uint8_t getConfiguration ( int i, int j ); //Method for reporting
the current configuration
    void setupRules(); //Method for initializing the default rules

    uint8_t *SBox; //Pointer to SBox (Normal Margolis), or Key (Single
Margolis) to be used as the CA Map
    uint8_t *invSBox; //Pointer to InvSBox (Normal Margolis), or NULL
(Single Margolis) to be used as the CA Map
    bool rules[16][6]; //Rules Matrix 16 configurations possible for
each

    bool evenodd; //Variable for storing current even or odd for
Margolis Neighborhood
    int thresh; //Threshhold value for determining if a cell is on or
off
    int height; //Height of the CA Map (for 2d indexing into a 1d map)
    int width; //Width of the CA Map (for 2d indexing into a 1d map)
};

class margolisSingle : public margolis //Single Margolis class for use
with Key array
{
public:
    void swap ( int x1, int y1, int x2, int y2 ); //Redefined swap
Method that doesn't attempt to mix an inverse box
```

```
    void setParams ( uint8_t *keyBoxz, int widthz, int heightz, int
threshz ); //Method for setting parameters for single Margolis
};

#include "margolis.cpp"

#endif
```

B4. Margolis Class Source File

```
#ifndef MARGOLIS_CPP
#define MARGOLIS_CPP

#include "margolis.h"

margolis::~margolis()
{} //Unused Default Destructor

margolis::margolis()
{} //Unused Default Constructor

void margolis::setupRules()
{
    /*Setup the BounceGas Rules for the Automata and initialize
EvenOdd*/

    evenodd = true;

    for ( int i = 0;i < 16;i++ ) for ( int j = 0;j < 6;j++ )
rules[i][j] = false;
    rules[1][0] = true;
    rules[2][1] = true;
    rules[4][1] = true;
    rules[6][2] = true;
    rules[6][3] = true;
    rules[7][0] = true;
    rules[8][0] = true;
    rules[9][2] = true;
    rules[9][3] = true;
    rules[11][1] = true;
    rules[13][1] = true;
    rules[14][0] = true;
}

void margolis::setParams ( uint8_t *sboxz, uint8_t *invsboxz, int
widthz, int heightz, int threshz )
{
    /*Set the Parameters and S-Box pointers for the S-box modifying
CA*/

    SBox = sboxz; //Set S-Box Pointer
    invSBox = invsboxz; //Set Inv S-Box Pointer
    width = widthz; //Set Width
    height = heightz; //Set Height
    thresh = threshz; //Set CA Threshold
```

```
    setupRules(); //Call Setup Rules
}

void margolisSingle::setParams ( uint8_t *keyBoxz, int widthz, int
heightz, int threshz )
{
    /*Set the Parameters and Key Array pointers for the Key modifying
CA*/

    SBox = keyBoxz; //Set Key Array pointer
    width = widthz; //Set Width
    height = heightz; //Set Height
    thresh = threshz; //Set CA Threshold

    setupRules(); //Call Setup Rules
}

void margolisSingle::swap ( int x1, int y1, int x2, int y2 )
{
    /*Swap method for Key-Modifying CA*/

    uint8_t temp = getCell ( x1, y1 ); //Swap Key Entries
    getCell ( x1, y1 ) = getCell ( x2, y2 );
    getCell ( x2, y2 ) = temp;
}

void margolis::swap ( int x1, int y1, int x2, int y2 )
{
    /*Swap method for S-Box modifying CA*/

    uint8_t temp = getCellInv ( x1, y1 ); //Swap Inv S-Box Entries
    getCellInv ( x1, y1 ) = getCellInv ( x2, y2 );
    getCellInv ( x2, y2 ) = temp;

    temp = getCell ( x1, y1 ); //Swap S-Box Entries
    getCell ( x1, y1 ) = getCell ( x2, y2 );
    getCell ( x2, y2 ) = temp;
}

void margolis::doTransition ( int x, int y, uint8_t currState )
{
    /*Apply the transition rules to the current parition
    based on its current state*/

    for ( int i = 0;i < 6;i++ ) //Check all Six Swaps
    {
        if ( rules[currState][i] ) //If swap is to be performed
        {
            switch ( i ) //Determine Swap and Do it
            {
            case 0:
                swapDiag1 ( x, y );
                break;
            case 1:
                swapDiag2 ( x, y );
                break;
            case 2:
```

106

```
                    swapHoriz1 ( x, y );
                    break;
                case 3:
                    swapHoriz2 ( x, y );
                    break;
                case 4:
                    swapVert1 ( x, y );
                    break;
                case 5:
                    swapVert2 ( x, y );
                    break;
            }
        }
    }
}

/*GetConfiguration adds up the the values of the cells in the partition
to a number between 0 and 15*/
#define getConfiguration(i,j) ((getCell(i+1,j+1)>thresh?8:0) +
(getCell(i,j+1)>thresh?4:0) + (getCell(i+1,j)>thresh?2:0) +
(getCell(i,j)>thresh?1:0))

void margolis::doGenerations ( int numGens )
{
    /*Method for performing a number of generations*/

    for ( int k = 0; k < numGens; k++ ) //For all the generations
    {
        evenodd ^= 1; //Invert the EvenOdd Boolean

        for ( int i = evenodd; i < height; i += 2 ) //Start from
EvenOdd and count by 2s
        {

            for ( int j = evenodd; j < width; j += 2 ) //Start from
EvenOdd and count by 2s
            {
                doTransition ( i, j, getConfiguration ( i, j ) ); //Do
the transition for the configuration of the partition
            }
        }
    }
}

#endif
```

B5. Tests Class Header File

```
#ifndef TESTS_H
#define TESTS_H

#include <iostream>
#include <fstream>
#include <cassert>
#include <sys/stat.h>
#include <math.h>
```

```
using std::cout;
using std::endl;
using std::hex;
using std::dec;
using std::cin;
using std::ifstream;
using std::ofstream;
using std::ios;

#define BIT(n) (1 << (n))

class tests //Class for performing various analysis
{

public:
    tests ( uint8_t *sboxz, uint8_t *invsboxz ); //Constructor takes an
SBox and invSBox pointer
    ~tests(); //Default Destructor

    void testInvertibility(); //Verifies invertibility between SBox and
invSBox
    void testNonLinearity ( bool verbose ); //Reports the Non-Linearity
measures of the SBox
    void testBitChanges ( bool verbose ); //Counts the number of bit
changes between sbox inputs and their outputs
    void BitARFWT(); //Performs Walsh Hadamard Transform for Non-
Linearity Measure
    void testDataHistogram ( bool verbose ); //Creates a data-histogram
file based on an input file
    void testMidPoints ( bool verbose ); //Reports the MidPoints of the
On and Off values in the CA SBox
    void testDistanceToCenter ( bool verbose ); //Reports the Distance
to the Center of MidPoints of the On and Off values in the CA SBox
    void testAvalanche ( bool verbose ); //Reports the effects of
modifying single bits in SBox input data
    void testAll ( bool verbose ); //Performs all tests
    void testEntropy ( bool verbose ); //Reports the Entropy of two
input files and their conditional entropy
    void printSbox(); //Prints out the SBox

private:
    uint8_t *sbox; //Pointer to SBox
    uint8_t *invsbox; //Pointer to InvSBox

    int bits[256]; //Walsh-Hadamard Transform Bits Array
};

#include "tests.cpp"

#endif
```

B6. Tests Class Source File

```
#ifndef TESTS_CPP
#define TESTS_CPP
```

```cpp
#include "tests.h"

tests::tests ( uint8_t *sboxz, uint8_t *invsboxz )
{
    /*Initialize the Class by setting the pointers*/
    sbox = sboxz;
    invsbox = invsboxz;
}

tests::~tests()
{} //Unused default destructor

void tests::printSbox()
{
    /*Function for Printing the S-Box in a 16x16 table*/

    for ( int i = 0;i < 256;i++ )
    {
        cout << hex << ( uint16_t ) sbox[i] << "\t";

        if ( ( i + 1 ) % 16 == 0 ) cout << endl;
    }

    cout << dec;
}

void tests::testEntropy ( bool verbose )
{
    /*Function for reporting entropy of X, Y, and X|Y*/
    char strXFileName[33];
    char strYFileName[33];

    do
    {
        cout << "***XFileName Name: <=32 Chars****" << endl;
        cin.getline ( strXFileName, 33 );
    }

    while ( cin.gcount() == 1 );

    cout << "X: " << strXFileName << endl;

    do
    {
        cout << "***YFileName Name: <=32 Chars****" << endl;
        cin.getline ( strYFileName, 33 );
    }

    while ( cin.gcount() == 1 );

    cout << "Y: " << strYFileName << endl;

    ifstream X ( strXFileName, ios::in | ios::binary );
    ifstream Y ( strYFileName, ios::in | ios::binary );

    struct stat inputXStats;
```

```
    struct stat inputYStats;

    stat ( strXFileName, &inputXStats );
    stat ( strYFileName, &inputYStats );

    int inputXByteCount = inputXStats.st_size;
    int inputYByteCount = inputYStats.st_size;

    if ( inputXByteCount != inputYByteCount )
    {
        cout << "Files must be the same size! " << inputXByteCount << "
!= " << inputYByteCount << endl;
        return;
    }

    uint8_t tempDataX;

    uint8_t tempDataY;

    double probX[256];
    double probY[256];

    double jointProb[256][256];

    for ( int i = 0;i < 256;i++ )
    {
        probX[i] = 0.0;
        probY[i] = 0.0;

        for ( int j = 0;j < 256;j++ )
        {
            jointProb[i][j] = 0.0;
        }
    }

    for ( int i = 0;i < inputXByteCount;i++ )
    {
        X.read ( ( char* ) &tempDataX, sizeof ( tempDataX ) );
        probX[tempDataX]++;

        Y.read ( ( char* ) &tempDataY, sizeof ( tempDataY ) );
        probY[tempDataY]++;

        jointProb[tempDataX][tempDataY]++;
    }

    X.close();

    Y.close();

    for ( int i = 0;i < 256;i++ )
    {
        probX[i] /= inputXByteCount;

        if ( verbose ) cout << dec << "Xp(" << i << ") = " << probX[i]
<< endl;
```

```cpp
        probY[i] /= inputYByteCount;

        if ( verbose ) cout << dec << "Yp(" << i << ") = " << probY[i]
<< endl;

        for ( int j = 0;j < 256;j++ )
        {
            jointProb[i][j] /= inputXByteCount;
        }
    }

    double condEntropy = 0.0;

    for ( int i = 0;i < 256;i++ )
    {
        for ( int j = 0;j < 256;j++ )
        {
            if ( ( jointProb[i][j] != 0.0 ) && ( probY[i] != 0.0 ) )
            {
                condEntropy += jointProb[i][j] * log ( probY[i] /
jointProb[i][j] );
            }
        }
    }

    condEntropy /= log ( 2.0 );

    if ( condEntropy <= 0.0 ) condEntropy = 0.0;

    cout << "Cond Entropy = " << condEntropy << endl;

    double entropyX = 0.0;

    double entropyY = 0.0;

    for ( int i = 0;i < 256;i++ )
    {
        if ( probX[i] != 0.0 ) entropyX -= probX[i] * log ( probX[i] );

        if ( probY[i] != 0.0 ) entropyY -= probY[i] * log ( probY[i] );
    }

    entropyX /= log ( 2.0 );

    entropyY /= log ( 2.0 );
    cout << "EntropyX = " << entropyX << endl;
    cout << "EntropyY = " << entropyY << endl;
}

void tests::testAll ( bool verbose )
{
    testMidPoints ( verbose );
    testDistanceToCenter ( verbose );
    testNonLinearity ( verbose );
    testAvalanche ( verbose );
    testBitChanges ( verbose );
    testInvertibility();
```

```
}

void tests::testDistanceToCenter ( bool verbose )
{
    double midPointTotalXT = 0;
    double midPointTotalYT = 0;
    double midPointCountT = 0;
    double midPointTotalXF = 0;
    double midPointTotalYF = 0;
    double midPointCountF = 0;

    for ( int i = 0;i < 16;i++ )
    {
        for ( int j = 0;j < 16;j++ )
        {
            if ( sbox[i*16+j] > 0x7f )
            {
                midPointCountT++;
                midPointTotalXT += j;
                midPointTotalYT += i;
            }

            else
            {
                midPointCountF++;
                midPointTotalXF += j;
                midPointTotalYF += i;
            }
        }
    }

    double distanceT = sqrt ( pow ( midPointTotalXT / midPointCountT -
7, 2 ) + pow ( midPointTotalYT / midPointCountT - 7, 2 ) );

    double distanceF = sqrt ( pow ( midPointTotalXF / midPointCountF -
7, 2 ) + pow ( midPointTotalXF / midPointCountF - 7, 2 ) );
    cout << "1's Distance from MidPoint = " << distanceT << " 0's
Distance from MidPoint = " << distanceF << endl;
//cout << distanceT << ";" << distanceF << endl;
}

void tests::testMidPoints ( bool verbose )
{
    double midPointTotalXT = 0;
    double midPointTotalYT = 0;
    double midPointCountT = 0;
    double midPointTotalXF = 0;
    double midPointTotalYF = 0;
    double midPointCountF = 0;

    for ( int i = 0;i < 16;i++ )
    {
        for ( int j = 0;j < 16;j++ )
        {
            if ( sbox[i*16+j] > 0x7f )
            {
                midPointCountT++;
```

```
                midPointTotalXT += j;
                midPointTotalYT += i;
            }

            else
            {
                midPointCountF++;
                midPointTotalXF += j;
                midPointTotalYF += i;
            }
        }
    }

    cout << "MidPoint for 1s: (" << midPointTotalXT / midPointCountT <<
"," << midPointTotalYT / midPointCountT << ") ";

    cout << "MidPoint for 0s: (" << midPointTotalXF / midPointCountF <<
"," << midPointTotalYF / midPointCountF << ")" << endl;
//cout << midPointTotalXT/midPointCountT << ";" <<
midPointTotalYT/midPointCountT << ";" << midPointTotalXF/midPointCountF
<< ";" << midPointTotalYF/midPointCountF << endl;
}

void tests::testInvertibility()
{
    for ( int i = 0; i < 256; i++ )
    {
        if ( invsbox[sbox[i]] != i )
        {
            cout << "Wrong value at " << i << endl;
        }

        else
        {
            if ( i == 255 ) cout << "OK all lookups succeeded" << endl;
        }
    }
}

void tests::BitARFWT()
{
    /*Walsh-Hadarman Transform Code from: www.ciphersbyritter.com*/
    int el1 = 0;
    int el2 = 0;
    int stradwid = 1;
    int bitARLast = 255;
    int blocks = 255;

    while ( stradwid != 0 )
    {
        el1 = 0;
        blocks >>= 1;

        for ( int block = 0; block <= blocks; block++ )
        {
            el2 = el1 + stradwid;
```

```cpp
            for ( int pair = 0; pair < stradwid; pair++ )
            {
                int a = bits[ el1 ];
                int b = bits[ el2 ];
                bits[ el1 ] = a + b;
                bits[ el2 ] = a - b;
                el1++;
                el2++;
            }

            el1 = el2;
        }

        stradwid = ( stradwid + stradwid ) & bitARLast;
    }
}

void tests::testNonLinearity ( bool verbose )
{
    int minNL = 0;

    for ( int i = 0;i < 8;i++ )
    {
        for ( int j = 0;j < 256;j++ )
        {
            bits[j] = ( sbox[j] & BIT ( i ) ) == 0 ? 0 : 1;
        }

        BitARFWT();

        for ( int i = 1;i < 256;i++ )
        {
            if ( abs ( bits[i] ) > minNL ) minNL = abs ( bits[i] );
        }
    }

    minNL = 128 - minNL;

    cout << "MinNL = " << dec << minNL << endl;
}

void tests::testDataHistogram ( bool verbose )
{
    char inFile[128];
    char outFile[128];

    do
    {
        cout << "Enter the data filename: " << endl;
        cin.getline ( inFile, 128 );
    }

    while ( cin.gcount() == 1 );

    do
    {
        cout << "Enter the results filename: " << endl;
```

```cpp
            cin.getline ( outFile, 128 );
        }

        while ( cin.gcount() == 1 );

        uint64_t Histogram[256];

        for ( int i = 0;i < 256;i++ ) Histogram[i] = 0;

        int n;

        struct stat results;

        n = stat ( inFile, &results );

        ifstream data ( inFile, ios::in | ios::binary );

        assert ( data );

        ofstream resultsFile ( outFile, ios::out );

        int c = 0;

        for ( int i = 0;i < results.st_size;i++ )
        {
            data.read ( ( char* ) &c, 1 );
            Histogram[c]++;
        }

        data.close();

        for ( int i = 0;i < 256;i++ )
        {
            resultsFile << hex << i << "," << dec << Histogram[i] << endl;

            if ( verbose ) cout << dec << Histogram[i] << "\t";

            if ( verbose ) if ( ( i + 1 ) % 16 == 0 ) cout << endl;
        }

        resultsFile << "Total Bytes: " << results.st_size << endl;

        if ( verbose ) cout << "Total Bytes: " << results.st_size << endl;

        resultsFile.close();
}

void tests::testBitChanges ( bool verbose )
{
    unsigned int totalBitChanges = 0;
    int count = 0;

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

        for ( int j = 0;j < 8;j++ )
```

```
        {
            if ( ( i&BIT ( j ) ) != ( sbox[i]&BIT ( j ) ) ) count++;

        }

        totalBitChanges += count;

        if ( verbose ) cout << "BitChange count for entry " << dec << i
<< " = " << count << "/" << ( 8 ) << " = " << count / ( float ) 8 <<
endl;
    }

    cout << "AvgBitChanges = " << totalBitChanges << "/" << ( 256*8 )
<< " = " << totalBitChanges / ( float ) ( 256*8 ) << endl;
}

void tests::testAvalanche ( bool verbose )
{
    /*Tests the avalanche effect by counting bit changes for single bit
changes in the input for all 256 entries*/
    unsigned int totalAvalanche = 0;
    int count = 0;
    int modified = 0;

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

        for ( int j = 0;j < 8;j++ )
        {
            modified = i ^ BIT ( j );

            for ( int k = 0;k < 8;k++ )
            {
                if ( ( sbox[i]&BIT ( k ) ) != ( sbox[modified]&BIT ( k
) ) ) count++;
            }
        }

        if ( verbose ) cout << "Avalanche for " << i << " = " << count
<< " / " << ( 8*8 ) << " = " << count / ( float ) ( 8*8 ) << endl;

        totalAvalanche += count;
    }

    cout << "Overall Avalanche = " << totalAvalanche << " / " << (
8*8*256 ) << " = " << totalAvalanche / ( float ) ( 8*8*256 ) << endl;
}

#endif
```

B7. Main Class Source File

```
#include <iostream>
```

116

```
#include <math.h>
#include <time.h>

#include "galois.h"

int main ( int argc, char *argv[] )
{
    srand ( time ( 0 ) ); //Initialize the random seed to the time

    galois *newgalois = new galois(); //Create a blocking instance of
galois

    system ( "PAUSE" ); //Pause then exit
    return 0;
}
```

APPENDIX C – Modular Test Case Sources and Test Results

C1. Galois Table Generation Test Source

```
#include <iostream>
#include <fstream>
#include <stdint.h>
#include <stdlib.h>

using namespace std;

uint8_t ltable[256];
uint8_t etable[256];
uint8_t gmulInv[256];
uint8_t SBox[256];
uint8_t SBoxInv[256];
uint8_t seed;
uint8_t expandKey[240];

const uint8_t generators[128] = {
0x03, 0x05, 0x06, 0x09, 0x0b, 0x0e, 0x11, 0x12, 0x13, 0x14, 0x17, 0x18,
0x19, 0x1a, 0x1c, 0x1e, 0x1f, 0x21, 0x22, 0x23, 0x27, 0x28, 0x2a, 0x2c,
0x30, 0x31, 0x3c, 0x3e, 0x3f, 0x41, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4b,
0x4c, 0x4e, 0x4f, 0x52, 0x54, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b, 0x5f,
0x64, 0x65, 0x68, 0x69, 0x6d, 0x6e, 0x70, 0x71, 0x76, 0x77, 0x79, 0x7a,
0x7b, 0x7e, 0x81, 0x84, 0x86, 0x87, 0x88, 0x8a, 0x8e, 0x8f, 0x90, 0x93,
0x95, 0x96, 0x98, 0x99, 0x9b, 0x9d, 0xa0, 0xa4, 0xa5, 0xa6, 0xa7, 0xa9,
0xaa, 0xac, 0xad, 0xb2, 0xb4, 0xb7, 0xb8, 0xb9, 0xba, 0xbe, 0xbf, 0xc0,
0xc1, 0xc4, 0xc8, 0xc9, 0xce, 0xcf, 0xd0, 0xd6, 0xd7, 0xda, 0xdc, 0xdd,
0xde, 0xe2, 0xe3, 0xe5, 0xe6, 0xe7, 0xe9, 0xea, 0xeb, 0xee, 0xf0, 0xf1,
0xf4, 0xf5, 0xf6, 0xf8, 0xfb, 0xfd, 0xfe, 0xff
                                };
```

```c
uint8_t ROTL(uint8_t x, uint8_t s)
{
    return (uint8_t) (((x)<<(s&(8-1))) | ((x)>>(8-(s&(8-1)))));
}

uint8_t ROTR(uint8_t x, uint8_t s)
{
    return (uint8_t) (((x)>>(s&(8-1))) | ((x)<<(8-(s&(8-1)))));
}

uint8_t gadd(uint8_t a, uint8_t b)
{
    return a ^ b;
}
uint8_t gsub(uint8_t a, uint8_t b)
{
    return a ^ b;
}

uint8_t gmul(uint8_t a, uint8_t b)
{
    uint8_t p = 0;
    uint8_t counter;
    uint8_t hi_bit_set;
    for (counter = 0; counter < 8; counter++)
    {
        if ((b & 1) == 1)
            p ^= a;
        hi_bit_set = (a & 0x80);
        a <<= 1;
        if (hi_bit_set == 0x80)
            a ^= 0x1b;
        b >>= 1;
    }
    return p;
}

void generateTables(uint8_t seed)
{
    etable[0] = 1;
    etable[255] = 1;
    for (int c=1;c<256;c++)
    {
        etable[c] = gmul(etable[c-1],seed);
        ltable[etable[c]] = c;
    }
    etable[255] = 1;
}

uint8_t gmulLookup(uint8_t a, uint8_t b)
{
    int s;
    int q;
    int z = 0;
    s = ltable[a] + ltable[b];
    s %= 255;
    s = etable[s];
```

```
        q = s;
        if (a == 0)
        {
            s = z;
        }
        else
        {
            s = q;
        }
        if (b == 0)
        {
            s = z;
        }
        else
        {
            q = z;
        }
        return s;
}

uint8_t gmulInverse(uint8_t in)
{
    if (in == 0)
        return 0;
    else
        return etable[(255 - ltable[in])];
}

void generateGmulInverse()
{
    gmulInv[0] = 0;
    for (int c=1;c<256;c++)
    {
        gmulInv[c] = gmulInverse(c);
    }
}

uint8_t sub(uint8_t in)
{
    uint8_t c, s, x;
    s = x = gmulInv[in];
    for (c = 0; c < 4; c++)
    {
        s = (s << 1) | (s >> 7);
        x ^= s;
    }
    x ^= seed;
    return x;
}

void generateSBoxen()
{
    for (int c=0;c<256;c++)
    {
        SBox[c] = sub(c);
        SBoxInv[SBox[c]] = c;
    }
```

```cpp
}

void printTables()
{
    cout << "Log Table" << endl;
    for (int i=0;i<256;i++)
    {
        cout << hex << (uint16_t) ltable[i] << "\t";
        if ((i+1)%16 == 0) cout << endl;
    }

    cout << endl << endl << endl;

    cout << "Anti-Log Table" << endl;
    for (int i=0;i<256;i++)
    {
        cout << hex << (uint16_t) etable[i] << "\t";
        if ((i+1)%16 == 0) cout << endl;
    }

    cout << endl << endl << endl;

    cout << "GMul Inverse Table" << endl;
    for (int i=0;i<256;i++)
    {
        cout << hex << (uint16_t) gmulInv[i] << "\t";
        if ((i+1)%16 == 0) cout << endl;
    }

    cout << endl << endl << endl;

    cout << "SBox Table" << endl;
    for (int i=0;i<256;i++)
    {
        cout << hex << (uint16_t) SBox[i] << "\t";
        if ((i+1)%16 == 0) cout << endl;
    }

    cout << endl << endl << endl;

    cout << "Inverse SBox Table" << endl;
    for (int i=0;i<256;i++)
    {
        cout << hex << (uint16_t) SBoxInv[i] << "\t";
        if ((i+1)%16 == 0) cout << endl;
    }

    cout << endl << endl << endl;

    cout << "Key Table" << endl;
    for (int i=0;i<240;i++)
    {
        cout << hex << (uint16_t) expandKey[i] << "\t";
        if ((i+1)%16 == 0) cout << endl;
    }
}
```

```cpp
void resetTables()
{
    for (int i=0;i<256;i++)
    {
        ltable[i] = etable[i] = gmulInv[i] = SBox[i] = SBoxInv[i] = 0;
    }
}

bool checkInvertibility()
{
    for (int i = 0; i < 256; i++)
    {
        if (SBoxInv[SBox[i]] != i)
        {
            cout << "Wrong value at " << i;

        }
        else
        {
            if (i == 255) cout << "OK all lookups succeeded" << endl;
        }
    }
}

void gmixColumn(uint8_t *r)
{
    uint8_t a[4];
    uint8_t c;
    for (c=0;c<4;c++)
    {
        a[c] = r[c];
    }
    r[0] = gmul(a[0],2) ^ gmul(a[3],1) ^ gmul(a[2],1) ^ gmul(a[1],3);
    r[1] = gmul(a[1],2) ^ gmul(a[0],1) ^ gmul(a[3],1) ^ gmul(a[2],3);
    r[2] = gmul(a[2],2) ^ gmul(a[1],1) ^ gmul(a[0],1) ^ gmul(a[3],3);
    r[3] = gmul(a[3],2) ^ gmul(a[2],1) ^ gmul(a[1],1) ^ gmul(a[0],3);
}

void invGmixColumn(uint8_t *r)
{
    uint8_t a[4];
    uint8_t c;
    for (c=0;c<4;c++)
    {
        a[c] = r[c];
    }

    r[0] = gmul(a[0],14) ^ gmul(a[3],9) ^ gmul(a[2],13) ^
gmul(a[1],11);
    r[1] = gmul(a[1],14) ^ gmul(a[0],9) ^ gmul(a[3],13) ^
gmul(a[2],11);
    r[2] = gmul(a[2],14) ^ gmul(a[1],9) ^ gmul(a[0],13) ^
gmul(a[3],11);
    r[3] = gmul(a[3],14) ^ gmul(a[2],9) ^ gmul(a[1],13) ^
gmul(a[0],11);
}
```

```
void rotate(uint8_t *in)
{
    uint8_t a,c;
    a = in[0];
    for (c=0;c<3;c++) in[c] = in[c + 1];
    in[3] = a;
    return;
}

uint8_t rcon(uint8_t in)
{
    uint8_t c=1;
    if (in == 0) return 0;
    while (in != 1)
    {
        c = gmul(c,2);
        in--;
    }
    return c;
}

void schedule_core(uint8_t *in, uint8_t i)
{
    char a;
    rotate(in);
    for (int a = 0; a < 4; a++)
        in[a] = SBox[in[a]];
    in[0] ^= rcon(i);
}

void expand_key(uint8_t *in)
{
    uint8_t t[4];
    uint8_t c = 32;
    uint8_t a;
    uint8_t i = 1;
    while (c < 240)
    {
        for (a = 0; a < 4; a++) t[a] = in[a + c - 4];
        if (c % 32 == 0)
        {
            schedule_core(t,i);
            i++;
        }

        if (c % 32 == 16)
        {
            for (a = 0; a < 4; a++) t[a] = SBox[t[a]];
        }
        for (a = 0; a < 4; a++)
        {
            in[c] = in[c - 32] ^ t[a];
            c++;
        }
    }
}
```

```cpp
void printKey()
{
    for (int i=0;i<240;i++)
    {
        cout << "i=" << i << " " << hex << (uint16_t) expandKey[i] <<
endl;
    }
}

int main()
{
    uint8_t test[] = { 0xdb, 0x13, 0x53, 0x45 };

    for (int i=0;i<32;i++) expandKey[i] = 0xff;

    //Verify Generator Tables
    //verifyExe();
    for (int i= 0;i<128;i++)
    {
        seed = generators[i];
        cout << "Generator = " << hex << (uint16_t) generators[i] <<
endl;
        //Zero tables for a new galois genertor
        resetTables();
        //Generate Log/Exponentiation Tables
        generateTables(generators[i]);
        //Generate 1/x Tables
        generateGmulInverse();
        //Generate SBox based on galois generator.
        generateSBoxen();
        //Check SBox to SBoxInv
        checkInvertibility();
        //Check GMix
        //gmixColumn(test);
        //for(int i=0;i<4;i++) cout << hex << (uint16_t) test[i] << "
";
        //Check InvGMix
        //invGmixColumn(test);
        //for(int i=0;i<4;i++) cout << hex << (uint16_t) test[i] << "
";
        expand_key(expandKey);
        //printTables();
        cout << "Generator " << i << ": " << endl;
        system("pause");
    }
    //system("pause");

    return 0;
}
```

C2. Border Wrapping Test

```c
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 32
```

124

```
#define WIDTH  32

int main()
{
    for (int i=0;i<HEIGHT;i++)
    {
        for (int j=0;j<WIDTH;j++)
        {
            printf("Left side for i = %d = %d\n", i, (HEIGHT + (i-
1))%HEIGHT);
            printf("Right side for i = %d = %d\n", i, (HEIGHT +
(i+1))%HEIGHT);
            printf("Bottom side for i = %d = %d\n", j, (WIDTH + (j-
1))%WIDTH);
            printf("Top side for i = %d = %d\n", j, (WIDTH +
(j+1))%WIDTH);
        }
        system("pause");
    }
    return 0;
}
```

C3. Non-Linearity Test

```
#include <iostream>
#include <math.h>

using std::cout;
using std::endl;
using std::hex;

int test[256] =
    {0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
0x2b, 0xfe, 0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47,
0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7, 0xfd, 0x93,
0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31,
0x15, 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80,
0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a,
0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00,
0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58,
0xcf, 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02,
0x7f, 0x50, 0x3c, 0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38,
0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c, 0x13,
0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8,
0x14, 0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24,
0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, 0xe7, 0xc8, 0x37,
0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae,
0x08, 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74,
0x1f, 0x4b, 0xbd, 0x8b, 0x8a, 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6,
0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98,
0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28,
0xdf, 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d,
0x0f, 0xb0, 0x54, 0xbb, 0x16};

int bits[256];
```

```
#define BIT(n) (1 << (n))

void BitARFWT()
{ /*www.ciphersbyritter.com*/
    int el1 = 0;
    int el2 = 0;
    int stradwid = 1;
    int bitARLast = 255;
    int blocks = 255;

    while (stradwid != 0)
    {
        el1 = 0;
        blocks >>= 1;
        for (int block=0; block <= blocks; block++)
        {
            el2 = el1 + stradwid;
            for (int pair=0; pair < stradwid; pair++)
            {
                int a = bits[ el1 ];
                int b = bits[ el2 ];
                bits[ el1 ] = a + b;
                bits[ el2 ] = a - b;
                el1++;
                el2++;
            }
            el1 = el2;
        }
        stradwid = (stradwid + stradwid) & bitARLast;
    }
}

void getNonLinearity()
{
    long unsigned int avgNL=0;
    int minNL = 0;
    int maxNL = 256;

    for (int i=0;i<8;i++)
    {
        for (int j=0;j<256;j++)
        {
            bits[j] = (test[j]&BIT(i))==0?0:1;
        }
        BitARFWT();
        for (int i=1;i<256;i++)
        {
            if (abs(bits[i]) > minNL) minNL = abs(bits[i]);
            if (abs(bits[i]) < maxNL) maxNL = abs(bits[i]);
            avgNL += 128 - abs(bits[i]);
        }
    }

    avgNL /= (255*8);
    minNL = 128 - minNL;
    maxNL = 128 - maxNL;
```

```
    cout << "AvgNL = " << avgNL << endl;
    cout << "MinNL = " << minNL << endl;
    cout << "MaxNL = " << maxNL << endl;
}


int main()
{
    getNonLinearity();

    system("pause");
    return 0;
}
```

C4. ShiftRow Test

```
#include <iostream>
#include <stdlib.h>

using namespace std;


void rotate(uint8_t *in)
{
    uint8_t a,c;
    a = in[0];
    for (c=0;c<7;c++) in[c] = in[c + 1];
    in[7] = a;
    return;
}

void shiftRows(uint8_t in[4][8])
{
    for (int i=1;i<4;i++)
    {
        for (int j=0;j<i;j++) rotate(in[i]);
    }
}

void unshiftRows(uint8_t in[4][8])
{
    for (int i=1;i<4;i++)
    {
        for (int j=4-i;j>0;j--) rotate(in[i]);
    }
}

void printOut(uint8_t in[4][8])
{
    for (int i=0;i<4;i++)
    {
        for (int j=0;j<8;j++)
        {
            if(in[i][j]<16)cout << "0";
            else cout << ""; cout << hex << (uint16_t) in[i][j] << " ";
```

```
        }
        cout << endl;
    }
    cout << endl << endl;
}

int main()
{
    for (int i=1;i<4;i++)
    {
        for (int j=0;j<i;j++) cout << "Shifting " << i << endl;
    }

    for (int i=1;i<4;i++)
    {
        for (int j=4-i;j>0;j--) cout << "UnShifting " << i << endl;
    }

    uint8_t blah[4][8] =
{{0x00,0x08,0x10,0x18,0x20,0x28,0x30,0x38},{0x40,0x48,0x50,0x58,0x60,0x
68,0x70,0x78},{0x80,0x88,0x90,0x98,0xA0,0xA8,0xB0,0xB8},{0xC0,0xC8,0xD0
,0xD8,0xE0,0xE8,0xF0,0xF8}};

    printOut(blah);
    shiftRows(blah);
    printOut(blah);
    unshiftRows(blah);
    printOut(blah);

    system("pause");
    return 0;
}
```

C5. Column Mix Test

```
#include <iostream>
#include <stdint.h>
#include <stdlib.h>

using namespace std;

uint8_t gmul(uint8_t a, uint8_t b)
{
    uint8_t p = 0;
    uint8_t counter;
    uint8_t hi_bit_set;
    for (counter = 0; counter < 8; counter++)
    {
        if ((b & 1) == 1)
            p ^= a;
        hi_bit_set = (a & 0x80);
        a <<= 1;
        if (hi_bit_set == 0x80)
            a ^= 0x1b;
        b >>= 1;
    }
```

```
    return p;
}

void gmixColumn(uint8_t *r)
{
    uint8_t a[4];
    uint8_t c;
    for (c=0;c<4;c++)
    {
        a[c] = r[c];
    }
    r[0] = gmul(a[0],2) ^ gmul(a[3],1) ^ gmul(a[2],1) ^ gmul(a[1],3);
    r[1] = gmul(a[1],2) ^ gmul(a[0],1) ^ gmul(a[3],1) ^ gmul(a[2],3);
    r[2] = gmul(a[2],2) ^ gmul(a[1],1) ^ gmul(a[0],1) ^ gmul(a[3],3);
    r[3] = gmul(a[3],2) ^ gmul(a[2],1) ^ gmul(a[1],1) ^ gmul(a[0],3);
}

void invGmixColumn(uint8_t *r)
{
    uint8_t a[4];
    uint8_t c;
    for (c=0;c<4;c++)
    {
        a[c] = r[c];
    }

    r[0] = gmul(a[0],14) ^ gmul(a[3],9) ^ gmul(a[2],13) ^
gmul(a[1],11);
    r[1] = gmul(a[1],14) ^ gmul(a[0],9) ^ gmul(a[3],13) ^
gmul(a[2],11);
    r[2] = gmul(a[2],14) ^ gmul(a[1],9) ^ gmul(a[0],13) ^
gmul(a[3],11);
    r[3] = gmul(a[3],14) ^ gmul(a[2],9) ^ gmul(a[1],13) ^
gmul(a[0],11);
}

void mixColumns(uint8_t in[4][8])
{
    uint8_t temp[4];
    for (int j=0;j<8;j++)
    {
        for (int i=0;i<4;i++)
        {
            temp[i] = in[i][j];
        }
        gmixColumn(temp);
        for (int i=0;i<4;i++)
        {
            in[i][j] = temp[i];
        }
    }
}

void unmixColumns(uint8_t in[4][8])
{
    uint8_t temp[4];
    for (int j=0;j<8;j++)
```

```
        {
            for (int i=0;i<4;i++)
            {
                temp[i] = in[i][j];
            }
            invGmixColumn(temp);
            for (int i=0;i<4;i++)
            {
                in[i][j] = temp[i];
            }
        }
}

void printOut(uint8_t in[4][8])
{
    for (int i=0;i<4;i++)
    {
        for (int j=0;j<8;j++)
        {
            if(in[i][j]<16)cout << "0";
            else cout << ""; cout << hex << (uint16_t) in[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl << endl;
}

int main()
{
    uint8_t blah[4][8] =
{{0x00,0x08,0x10,0x18,0x20,0x28,0x30,0x38},{0x40,0x48,0x50,0x58,0x60,0x
68,0x70,0x78},{0x80,0x88,0x90,0x98,0xA0,0xA8,0xB0,0xB8},{0xC0,0xC8,0xD0
,0xD8,0xE0,0xE8,0xF0,0xF8}};
    printOut(blah);
    mixColumns(blah);
    printOut(blah);
    unmixColumns(blah);
    printOut(blah);
    system("pause");
    return 0;
}
```

C6. Clock Drift IV Generation Test

```
#include <iostream>
#include <time.h>

using std::cout;
using std::endl;
using std::ios;
using std::hex;
using std::dec;

int main()
{
    uint8_t data[256*64];
```

```cpp
    clock_t start_tick;
    for (int x=0;x<256*64;x++)
    {
        data[x] = 0;
        start_tick = clock();
        while (clock() == start_tick)
        {
            data[x]++;
        }
    }

    for(int i=0;i<256*64;i++)
    {
        cout << i << "\t" << hex << (uint16_t) data[i] << endl;
    }

    uint8_t hist[256];
    for(int i=0;i<256;i++)
    {
        hist[i] = 0;
    }

    for(int i=0;i<256*64;i++)
    {
        hist[data[i]]++;
    }

    cout << endl << endl;

    for(int i=0;i<256;i++)
    {
        cout << dec << i << "," << dec << (uint16_t) hist[i] << endl;
    }

    system("pause");
    return 0;
}
```

<u>C7. Padding Test and Results</u>

```bash
#FileName: doTest.sh
#!/bin/bash
rm text.txt
cd /home/shambler/Thesis/Margalois
for((i=0;i<33;i++))
do
echo -n "A" >> text.txt;
./a.out < runScriptEncrypt > /dev/null;
./a.out < runScriptDecrypt > /dev/null;
ls -l *.txt;
md5sum *.txt;
done

#FileName: runScriptEncrypt
e
```

```
1
myPasswordKey
text.txt
ciph.txt
0
1
0
127
n

#FileName: runScriptDecrypt
d
1
myPasswordKey
newtext.txt
ciph.txt
0
1
0
127
n


#FileName: PaddingResults.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users  1 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users  1 2007-06-02 16:31 text.txt
4b5bc888e08b7c3241dfa18564749292  ciph.txt
7fc56270e7a70fa81a5935b72eacbe29  newtext.txt
7fc56270e7a70fa81a5935b72eacbe29  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users  2 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users  2 2007-06-02 16:31 text.txt
50fbadc7a5abc1f82b198b1f22a2599e  ciph.txt
3b98e2dffc6cb06a89dcb0d5c60a0206  newtext.txt
3b98e2dffc6cb06a89dcb0d5c60a0206  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users  3 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users  3 2007-06-02 16:31 text.txt
c62414a5cc9167f261d249176974335a  ciph.txt
e1faffb3e614e6c2fba74296962386b7  newtext.txt
e1faffb3e614e6c2fba74296962386b7  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users  4 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users  4 2007-06-02 16:31 text.txt
9467faad9af733e43226ad85b75ca9e8  ciph.txt
098890dde069e9abad63f19a0d9e1f32  newtext.txt
098890dde069e9abad63f19a0d9e1f32  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users  5 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users  5 2007-06-02 16:31 text.txt
7d30a0a505fb38f58e4468665c9bd7b6  ciph.txt
f6a6263167c92de8644ac998b3c4e4d1  newtext.txt
f6a6263167c92de8644ac998b3c4e4d1  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users  6 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users  6 2007-06-02 16:31 text.txt
```

```
374c4693420730ef3d655bf377d7681d  ciph.txt
36d04a9d74392c727b1a9bf97a7bcbac  newtext.txt
36d04a9d74392c727b1a9bf97a7bcbac  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users  7 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users  7 2007-06-02 16:31 text.txt
a781ab5862c1d70b272e46e2148777e3  ciph.txt
8430894cfeb54a3625f18fe24fce272e  newtext.txt
8430894cfeb54a3625f18fe24fce272e  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users  8 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users  8 2007-06-02 16:31 text.txt
081b3a9ba33e7e278b3b4b7c05cc4a7f  ciph.txt
aee9e38cb4d40ec2794542567539b4c8  newtext.txt
aee9e38cb4d40ec2794542567539b4c8  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users  9 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users  9 2007-06-02 16:31 text.txt
f3f5f53f5d9767fac37c52ab5ac4e3c3  ciph.txt
6c9395cacd317eed2777f669103b7181  newtext.txt
6c9395cacd317eed2777f669103b7181  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 10 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 10 2007-06-02 16:31 text.txt
8545b583d6ed814fe263d8b08469d576  ciph.txt
16c52c6e8326c071da771e66dc6e9e57  newtext.txt
16c52c6e8326c071da771e66dc6e9e57  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 11 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 11 2007-06-02 16:31 text.txt
8d9ea3b18f34da611ba25bca01eea1d2  ciph.txt
aae9ed2aebd46960a986cfb376bc1eca  newtext.txt
aae9ed2aebd46960a986cfb376bc1eca  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 12 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 12 2007-06-02 16:31 text.txt
568348987f6d18af058f6767a7ee8243  ciph.txt
02737e4e8c87d7466b623c1f844fdd71  newtext.txt
02737e4e8c87d7466b623c1f844fdd71  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 13 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 13 2007-06-02 16:31 text.txt
c1ba67d0dff94d5ce0d1cd911fbbd254  ciph.txt
a68c7b41f873e90566acec7c22f89824  newtext.txt
a68c7b41f873e90566acec7c22f89824  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 14 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 14 2007-06-02 16:31 text.txt
ee563345e901b0d0862b548b2e51b4f7  ciph.txt
74d8c66251bba513d7d317dd47f556ba  newtext.txt
74d8c66251bba513d7d317dd47f556ba  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 15 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 15 2007-06-02 16:31 text.txt
a47d0f1fc70e3fcde7a7c1ab86151a12  ciph.txt
409c94b762769ea5fb9384eb9bddf207  newtext.txt
409c94b762769ea5fb9384eb9bddf207  text.txt
```

```
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 16 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 16 2007-06-02 16:31 text.txt
87dd2d7909afdafc67a3462d69c53d79  ciph.txt
d8a73157ce10cd94a91c2079fc9a92c8  newtext.txt
d8a73157ce10cd94a91c2079fc9a92c8  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 17 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 17 2007-06-02 16:31 text.txt
f95074577aa85825d8f38b135c8164c3  ciph.txt
1105d53d33874fe294a18ee36398f2dc  newtext.txt
1105d53d33874fe294a18ee36398f2dc  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 18 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 18 2007-06-02 16:31 text.txt
665a94ca1a36b8f011fd2854ab8e5f36  ciph.txt
9fe125b6680b43a62953d4cc6f4e08bf  newtext.txt
9fe125b6680b43a62953d4cc6f4e08bf  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 19 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 19 2007-06-02 16:31 text.txt
0338ac039e3964b786829497914775d8  ciph.txt
7ae4d6728e33ff002bf67a2e5194ccb1  newtext.txt
7ae4d6728e33ff002bf67a2e5194ccb1  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 20 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 20 2007-06-02 16:31 text.txt
ac3269e385c3d02ca3e94daac9f2d199  ciph.txt
76d36e98f312e98ff908c8c82c8dd623  newtext.txt
76d36e98f312e98ff908c8c82c8dd623  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 21 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 21 2007-06-02 16:31 text.txt
825cd1e108204ae124920b635f0f0d88  ciph.txt
59f34ff3997b416f4f2dee1c9776c0cd  newtext.txt
59f34ff3997b416f4f2dee1c9776c0cd  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 22 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 22 2007-06-02 16:31 text.txt
b8222f08a01c2a4e9260c53ee65ad3f0  ciph.txt
8b4cc90d421780e7674e2a25db33b770  newtext.txt
8b4cc90d421780e7674e2a25db33b770  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:31 ciph.txt
-rw-r--r-- 1 shambler users 23 2007-06-02 16:31 newtext.txt
-rw-r--r-- 1 shambler users 23 2007-06-02 16:31 text.txt
d8abd5eab8bdbb5b3ce87c1b0855cc06  ciph.txt
38079371e04ce549db3e4d69bc96b3ad  newtext.txt
38079371e04ce549db3e4d69bc96b3ad  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users 24 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users 24 2007-06-02 16:31 text.txt
ab390cdb9460daf3d58f6fda3b8484d4  ciph.txt
c7c6abfa9cb508f7fc178d4045313a94  newtext.txt
c7c6abfa9cb508f7fc178d4045313a94  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users 25 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users 25 2007-06-02 16:32 text.txt
```

```
13194c6874dab92645c98b199f6d6a54  ciph.txt
1995da96cd16a48cebcbc08424f6f945  newtext.txt
1995da96cd16a48cebcbc08424f6f945  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users 26 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users 26 2007-06-02 16:32 text.txt
7224b7e5c0724b78f6edfff1b4c39ec1  ciph.txt
9894d0235313057edec272848ca193f3  newtext.txt
9894d0235313057edec272848ca193f3  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users 27 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users 27 2007-06-02 16:32 text.txt
c27e387e6794c203d40422b1375e7776  ciph.txt
878d9f8dea73b35e1d23570409b0a09d  newtext.txt
878d9f8dea73b35e1d23570409b0a09d  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users 28 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users 28 2007-06-02 16:32 text.txt
7b311aa89bf0a03bbea37ea1d900b305  ciph.txt
35ea99843da5ff0639992be381c5b77a  newtext.txt
35ea99843da5ff0639992be381c5b77a  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users 29 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users 29 2007-06-02 16:32 text.txt
3a4e3504c796e29c3630e2222f9b8fc5  ciph.txt
cf5205dc20fb05145e6d1fa08166e94e  newtext.txt
cf5205dc20fb05145e6d1fa08166e94e  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users 30 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users 30 2007-06-02 16:32 text.txt
182d0f7f9e69f23e3839600117ea2fb1  ciph.txt
a8a7d9c5e31058f15d25f18d7d65404a  newtext.txt
a8a7d9c5e31058f15d25f18d7d65404a  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users 31 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users 31 2007-06-02 16:32 text.txt
a1ab7389ad453f36e69bc79d4a65e26d  ciph.txt
d09170db213e1a1fdc5effd49fd34767  newtext.txt
d09170db213e1a1fdc5effd49fd34767  text.txt
-rw-r--r-- 1 shambler users 96 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users 32 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users 32 2007-06-02 16:32 text.txt
b5ccb9536c4b08f2a58eb4b75ce31125  ciph.txt
5216ddcc58e8dade5256075e77f642da  newtext.txt
5216ddcc58e8dade5256075e77f642da  text.txt
-rw-r--r-- 1 shambler users 128 2007-06-02 16:32 ciph.txt
-rw-r--r-- 1 shambler users  33 2007-06-02 16:32 newtext.txt
-rw-r--r-- 1 shambler users  33 2007-06-02 16:32 text.txt
3d6c9595b8402b350aab89ac8c09d18e  ciph.txt
eeda92ae5deb94f83a420113abf8db3e  newtext.txt
eeda92ae5deb94f83a420113abf8db3e  text.txt
```

C8. Time Analysis and Profiling Results

```
Generator Tables Verified! Check: 54 Compare: 54
Encrypt or Decrypt? (e,d)
```

135

```
Would you like to enter a 1)Passphrase or 2)Key?
***Input 32 Ascii Characters****
***TextFile Name: <=32 Chars****
***CiphFile Name: <=32 Chars****
Enter your galois generator number (0-127)
Enter the number of generations per block:
Enter the number of key generations per block:
Enter the threshold value for the Automata: (64-192) (127 RECOMMENDED)
Would you like to run (1) generations before key expansion? (y,n)
0.000089 InitCipherTime
0.631368 InitIVTime
17.200998 Encrypt/DecryptTime
Flat profile:


Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 57.76     21.76    21.76 13631800     0.00     0.00
galois::gmixColumn(unsigned char*)
 18.04     28.55     6.80  6071822     0.00     0.00
margolis::swap(int, int, int, int)
 14.58     34.05     5.49   262146     0.00     0.00
margolis::doGenerations(int)
  8.95     37.42     3.37   131075     0.00     0.00
galois::doRounds()
  0.24     37.51     0.09        1     0.09    37.57  galois::encrypt()
  0.16     37.57     0.06        1     0.06     0.06  galois::initIV()
  0.15     37.62     0.06
margolisSingle::swap(int, int, int, int)
  0.15     37.68     0.06
galois::generateSBoxes()
  0.00     37.68     0.00        4     0.00     0.00  void
std::fill<char*, int>(char*, char*, int const&)
  0.00     37.68     0.00        2     0.00     0.00
std::hex(std::ios_base&)
  0.00     37.68     0.00        1     0.00     0.00  global
constructors keyed to _ZN8margolis10setupRulesEv
  0.00     37.68     0.00        1     0.00     0.00
galois::expand_key()
  0.00     37.68     0.00        1     0.00     0.00
galois::initCipher()
  0.00     37.68     0.00        1     0.00     0.00
galois::textUserPrompt()
  0.00     37.68     0.00        1     0.00    37.57
galois::doEncryptDecrypt()
  0.00     37.68     0.00        1     0.00     0.00
galois::verifyGenerators()
  0.00     37.68     0.00        1     0.00     0.00
margolis::setParams(unsigned char*, unsigned char*, int, int, int)
  0.00     37.68     0.00        1     0.00     0.00
std::dec(std::ios_base&)


granularity: each sample hit covers 2 byte(s) for 0.03% of 37.68
seconds


index % time    self  children    called     name
```

```
                0.09    37.48        1/1
galois::doEncryptDecrypt() [3]
[1]     99.7    0.09    37.48        1           galois::encrypt() [1]
                3.37    21.76  131073/131075    galois::doRounds() [4]
                5.49     6.80  262146/262146
margolis::doGenerations(int) [6]
                0.06     0.00        1/1           galois::initIV() [8]
-------------------------------------------------
                                                  <spontaneous>
[2]     99.7    0.00    37.57                    main [2]
                0.00    37.57        1/1
galois::doEncryptDecrypt() [3]
                0.00     0.00        1/1
galois::verifyGenerators() [23]
                0.00     0.00        1/1
galois::textUserPrompt() [22]
                0.00     0.00        1/1           galois::initCipher()
[21]
-------------------------------------------------
                0.00    37.57        1/1           main [2]
[3]     99.7    0.00    37.57        1           galois::doEncryptDecrypt()
[3]
                0.09    37.48        1/1           galois::encrypt() [1]
-------------------------------------------------
                0.00     0.00        2/131075    galois::initIV() [8]
                3.37    21.76  131073/131075    galois::encrypt() [1]
[4]     66.7    3.37    21.76   131075          galois::doRounds() [4]
               21.76     0.00 13631800/13631800
galois::gmixColumn(unsigned char*) [5]
-------------------------------------------------
               21.76     0.00 13631800/13631800    galois::doRounds()
[4]
[5]     57.8   21.76     0.00 13631800
galois::gmixColumn(unsigned char*) [5]
-------------------------------------------------
                5.49     6.80  262146/262146    galois::encrypt() [1]
[6]     32.6    5.49     6.80   262146
margolis::doGenerations(int) [6]
                6.80     0.00 6071822/6071822    margolis::swap(int,
int, int, int) [7]
-------------------------------------------------
                6.80     0.00 6071822/6071822
margolis::doGenerations(int) [6]
[7]     18.0    6.80     0.00 6071822          margolis::swap(int, int,
int, int) [7]
-------------------------------------------------
                0.06     0.00        1/1           galois::encrypt() [1]
[8]      0.2    0.06     0.00        1         galois::initIV() [8]
                0.00     0.00        2/131075    galois::doRounds() [4]
-------------------------------------------------
                                                  <spontaneous>
[9]      0.1    0.06     0.00                    margolisSingle::swap(int,
int, int, int) [9]
-------------------------------------------------
                                                  <spontaneous>
[10]     0.1    0.06     0.00                    galois::generateSBoxes()
[10]
```

137

```
                -----------------------------------------------
                        0.00    0.00      4/4
galois::textUserPrompt() [22]
[17]     0.0    0.00    0.00      4         void std::fill<char*,
int>(char*, char*, int const&) [17]
                -----------------------------------------------
                        0.00    0.00      2/2
galois::verifyGenerators() [23]
[18]     0.0    0.00    0.00      2         std::hex(std::ios_base&)
[18]
                -----------------------------------------------
                        0.00    0.00      1/1          __do_global_ctors_aux
[80]
[19]     0.0    0.00    0.00      1         global constructors keyed
to _ZN8margolis10setupRulesEv [19]
                -----------------------------------------------
                        0.00    0.00      1/1          galois::initCipher()
[21]
[20]     0.0    0.00    0.00      1         galois::expand_key() [20]
                -----------------------------------------------
                        0.00    0.00      1/1          main [2]
[21]     0.0    0.00    0.00      1         galois::initCipher() [21]
                        0.00    0.00      1/1
margolis::setParams(unsigned char*, unsigned char*, int, int, int) [24]
                        0.00    0.00      1/1          galois::expand_key()
[20]
                -----------------------------------------------
                        0.00    0.00      1/1          main [2]
[22]     0.0    0.00    0.00      1         galois::textUserPrompt()
[22]
                        0.00    0.00      4/4          void std::fill<char*,
int>(char*, char*, int const&) [17]
                        0.00    0.00      1/1
std::dec(std::ios_base&) [25]
                -----------------------------------------------
                        0.00    0.00      1/1          main [2]
[23]     0.0    0.00    0.00      1         galois::verifyGenerators()
[23]
                        0.00    0.00      2/2
std::hex(std::ios_base&) [18]
                -----------------------------------------------
                        0.00    0.00      1/1          galois::initCipher()
[21]
[24]     0.0    0.00    0.00      1
margolis::setParams(unsigned char*, unsigned char*, int, int, int) [24]
                -----------------------------------------------
                        0.00    0.00      1/1
galois::textUserPrompt() [22]
[25]     0.0    0.00    0.00      1         std::dec(std::ios_base&)
[25]
                -----------------------------------------------
```

Index by function name

```
     [9] margolisSingle::swap(int, int, int, int) [3]
galois::doEncryptDecrypt() [7] margolis::swap(int, int, int, int)
  [20] galois::expand_key()   [23] galois::verifyGenerators() [24]
margolis::setParams(unsigned char*, unsigned char*, int, int, int)
   [5] galois::gmixColumn(unsigned char*) [8] galois::initIV() [25]
std::dec(std::ios_base&)
  [21] galois::initCipher()    [1] galois::encrypt()      [18]
std::hex(std::ios_base&)
  [10] galois::generateSBoxes() [4] galois::doRounds()    [17] void
std::fill<char*, int>(char*, char*, int const&)
Generator Tables Verified! Check: 54 Compare: 54
Encrypt or Decrypt? (e,d)
Would you like to enter a 1)Passphrase or 2)Key?
***Input 32 Ascii Characters****
***TextFile Name: <=32 Chars****
***CiphFile Name: <=32 Chars****
Enter your galois generator number (0-127)
Enter the number of generations per block:
Enter the number of key generations per block:
Enter the threshold value for the Automata: (64-192) (127 RECOMMENDED)
Would you like to run (1) generations before key expansion? (y,n)
0.000087 InitCipherTime
16.628243 Encrypt/DecryptTime
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 57.64     21.71    21.71 13631592    0.00     0.00
galois::invGmixColumn(unsigned char*)
 18.51     28.68     6.97  6071776    0.00     0.00
margolis::swap(int, int, int, int)
 14.39     34.10     5.42   262144    0.00     0.00
margolis::doGenerations(int)
  9.07     37.52     3.42   131073    0.00     0.00
galois::doInvRounds()
  0.17     37.58     0.07
galois::mixColumns()
  0.16     37.64     0.06        1    0.06    37.58  galois::decrypt()
  0.05     37.66     0.02
margolisSingle::swap(int, int, int, int)
  0.01     37.67     0.01
galois::invShiftRows()
  0.00     37.67     0.00        4    0.00     0.00  void
std::fill<char*, int>(char*, char*, int const&)
  0.00     37.67     0.00        2    0.00     0.00
std::hex(std::ios_base&)
  0.00     37.67     0.00        1    0.00     0.00  global
constructors keyed to _ZN8margolis10setupRulesEv
  0.00     37.67     0.00        1    0.00     0.00
galois::expand_key()
  0.00     37.67     0.00        1    0.00     0.00
galois::initCipher()
  0.00     37.67     0.00        1    0.00     0.00
galois::textUserPrompt()
  0.00     37.67     0.00        1    0.00    37.58
galois::doEncryptDecrypt()
```

```
   0.00     37.67     0.00        1      0.00      0.00
galois::verifyGenerators()
   0.00     37.67     0.00        1      0.00      0.00
margolis::setParams(unsigned char*, unsigned char*, int, int, int)
   0.00     37.67     0.00        1      0.00      0.00
std::dec(std::ios_base&)

granularity: each sample hit covers 2 byte(s) for 0.03% of 37.67
seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.8    0.00   37.58                      main [1]
                0.00   37.58       1/1
galois::doEncryptDecrypt() [2]
                0.00    0.00       1/1
galois::verifyGenerators() [23]
                0.00    0.00       1/1
galois::textUserPrompt() [22]
                0.00    0.00       1/1          galois::initCipher()
[21]
-----------------------------------------------
                0.00   37.58       1/1          main [1]
[2]     99.8    0.00   37.58       1          galois::doEncryptDecrypt()
[2]
                0.06   37.52       1/1          galois::decrypt() [3]
-----------------------------------------------
                0.06   37.52       1/1
galois::doEncryptDecrypt() [2]
[3]     99.8    0.06   37.52       1          galois::decrypt() [3]
                3.42   21.71  131073/131073    galois::doInvRounds()
[4]
                5.42    6.97  262144/262144
margolis::doGenerations(int) [6]
-----------------------------------------------
                3.42   21.71  131073/131073    galois::decrypt() [3]
[4]     66.7    3.42   21.71  131073          galois::doInvRounds() [4]
               21.71    0.00 13631592/13631592
galois::invGmixColumn(unsigned char*) [5]
-----------------------------------------------
               21.71    0.00 13631592/13631592
galois::doInvRounds() [4]
[5]     57.6   21.71    0.00 13631592
galois::invGmixColumn(unsigned char*) [5]
-----------------------------------------------
                5.42    6.97  262144/262144    galois::decrypt() [3]
[6]     32.9    5.42    6.97  262144
margolis::doGenerations(int) [6]
                6.97    0.00 6071776/6071776   margolis::swap(int,
int, int, int) [7]
-----------------------------------------------
                6.97    0.00 6071776/6071776
margolis::doGenerations(int) [6]
[7]     18.5    6.97    0.00 6071776          margolis::swap(int, int,
int, int) [7]
-----------------------------------------------
                                                 <spontaneous>
```

```
[8]      0.2    0.07    0.00                              galois::mixColumns() [8]
-------------------------------------------------
                                      <spontaneous>
[9]      0.1    0.02    0.00                              margolisSingle::swap(int,
int, int, int) [9]
-------------------------------------------------
                                      <spontaneous>
[10]     0.0    0.01    0.00                              galois::invShiftRows()
[10]
-------------------------------------------------
                0.00    0.00    4/4
galois::textUserPrompt() [22]
[17]     0.0    0.00    0.00       4         void std::fill<char*,
int>(char*, char*, int const&) [17]
-------------------------------------------------
                0.00    0.00    2/2
galois::verifyGenerators() [23]
[18]     0.0    0.00    0.00       2         std::hex(std::ios_base&)
[18]
-------------------------------------------------
                0.00    0.00    1/1          __do_global_ctors_aux
[80]
[19]     0.0    0.00    0.00       1         global constructors keyed
to _ZN8margolis10setupRulesEv [19]
-------------------------------------------------
                0.00    0.00    1/1          galois::initCipher()
[21]
[20]     0.0    0.00    0.00       1         galois::expand_key() [20]
-------------------------------------------------
                0.00    0.00    1/1          main [1]
[21]     0.0    0.00    0.00       1         galois::initCipher() [21]
                0.00    0.00    1/1
margolis::setParams(unsigned char*, unsigned char*, int, int, int) [24]
                0.00    0.00    1/1          galois::expand_key()
[20]
-------------------------------------------------
                0.00    0.00    1/1          main [1]
[22]     0.0    0.00    0.00       1         galois::textUserPrompt()
[22]
                0.00    0.00    4/4          void std::fill<char*,
int>(char*, char*, int const&) [17]
                0.00    0.00    1/1
std::dec(std::ios_base&) [25]
-------------------------------------------------
                0.00    0.00    1/1          main [1]
[23]     0.0    0.00    0.00       1         galois::verifyGenerators()
[23]
                0.00    0.00    2/2
std::hex(std::ios_base&) [18]
-------------------------------------------------
                0.00    0.00    1/1          galois::initCipher()
[21]
[24]     0.0    0.00    0.00       1
margolis::setParams(unsigned char*, unsigned char*, int, int, int) [24]
-------------------------------------------------
                0.00    0.00    1/1
galois::textUserPrompt() [22]
```

```
[25]     0.0    0.00    0.00        1          std::dec(std::ios_base&)
[25]
-----------------------------------------------
```

Index by function name

```
  [19] global constructors keyed to _ZN8margolis10setupRulesEv [10]
galois::invShiftRows() [6] margolis::doGenerations(int)
   [9] margolisSingle::swap(int, int, int, int) [5]
galois::invGmixColumn(unsigned char*) [7] margolis::swap(int, int, int,
int)
  [20] galois::expand_key()    [22] galois::textUserPrompt() [24]
margolis::setParams(unsigned char*, unsigned char*, int, int, int)
  [21] galois::initCipher()     [2] galois::doEncryptDecrypt() [25]
std::dec(std::ios_base&)
   [8] galois::mixColumns()    [23] galois::verifyGenerators() [18]
std::hex(std::ios_base&)
   [4] galois::doInvRounds()    [3] galois::decrypt()      [17] void
std::fill<char*, int>(char*, char*, int const&)
```

C9. Entropy Results

```
X: moby-dick.dat
Y: moby-dick-ciph-aes-sbox-only.dat
Cond Entropy = 0
EntropyX = 4.49714
EntropyY = 4.49714
X: moby-dick.dat
Y: moby-dick-ciph-AESCA1-sbox-only.dat
Cond Entropy = 4.35398
EntropyX = 4.49714
EntropyY = 7.98982
X: moby-dick.dat
Y: moby-dick-ciph-aesca2-sbox-only.dat
Cond Entropy = 4.4135
EntropyX = 4.49714
EntropyY = 7.99565
X: moby-dick.dat
Y: moby-dick-ciph-aesca5-sbox-only.dat
Cond Entropy = 4.4412
EntropyX = 4.49714
EntropyY = 7.99826
X: moby-dick.dat
Y: moby-dick-ciph-aesca10-sbox-only.dat
Cond Entropy = 4.44154
EntropyX = 4.49714
EntropyY = 7.99865
X: moby-dick.dat
Y: moby-dick-ciph-aesca50-sbox-only.dat
Cond Entropy = 4.46575
EntropyX = 4.49714
EntropyY = 7.99917
```

C10. Mauer's Results

```
/*
```

```
   ULISCAN.c   ---blocksize of 8

1 Oct 98
1 Dec 98
21 Dec 98
uliscan.c derived from ueli8.c
This version has // comments removed for Sun cc
This implements Ueli M Maurer's
"Universal Statistical Test for Random Bit Generators"
using L=8
Accepts a filename on the command line;
writes its results, with other info, to stdout.
Handles input file exhaustion gracefully.
Ref: J. Cryptology v 5 no 2, 1992 pp 89-105
also on the web somewhere, which is where I found it.

-David Honig
honig@xxxxxxxxxxx

Usage:
        ULISCAN filename
        outputs to stdout
*/

#define L 8
#define V (1<<L)
#define Q (10*V)
#define K (100    *Q)
#define MAXSAMP (Q + K)

#include <stdio.h>
#include <math.h>
#include <iostream>

int main(int argc, char **argv)
{
  FILE *fptr;
  int i,j;
  int b, c;
  int table[V];
  double sum = 0.0;
  int iproduct = 1;
  int run;

  printf("Uliscan 21 Dec 98 \nL=%d %d %d \n", L, V, MAXSAMP);

  if (argc < 2) {
    printf("Usage: Uliscan filename\n");
    exit(-1);
  } else {
    printf("Measuring file %s\n", argv[1]);
  }

  fptr = fopen(argv[1],"rb");

  if (fptr == NULL) {
    printf("Can't find %s\n", argv[1]);
```

143

```
    exit(-1);
}

for (i = 0; i < V; i++) {
  table[i] = 0;
}

for (i = 0; i < Q; i++) {
  b = fgetc(fptr);
  table[b] = i;
}

printf("Init done\n");

printf("Expected value for L=8 is 7.1836656\n");

run = 1;

while (run) {
  sum = 0.0;
  iproduct = 1;

  if (run)
    for (i = Q; run && i < Q + K; i++) {
      j = i;
      b = fgetc(fptr);

      if (b < 0)
        run = 0;

      if (run) {
        if (table[b] > j)
          j += K;

        sum += log((double)(j-table[b]));

        table[b] = i;
      }
    }

  if (!run)
    printf("Premature end of file; read %d blocks.\n", i - Q);

  sum = (sum/((double)(i - Q))) /  log(2.0);
  printf("%4.4f ", sum);

  for (i = 0; i < (int)(sum*8.0 + 0.50); i++)
    printf("-");

  printf("\n");

  /* refill initial table */
  if (0) {
    for (i = 0; i < Q; i++) {
      b = fgetc(fptr);
      if (b < 0) {
        run = 0;
```

```
        } else {
          table[b] = i;
        }
      }
    }
  }
}


Uliscan 21 Dec 98
L=8 256 258560
Measuring file moby-dick.dat
Init done
Expected value for L=8 is 7.1836656
3.8997 -----------------------------
3.9097 -----------------------------
3.8891 -----------------------------
Uliscan 21 Dec 98
L=8 256 258560
Measuring file moby-dick-ciph-aes-sbox-only.dat
Init done
Expected value for L=8 is 7.1836656
3.8997 -----------------------------
3.9097 -----------------------------
3.8891 -----------------------------
Uliscan 21 Dec 98
L=8 256 258560
Measuring file moby-dick-ciph-aesca1-sbox-only.dat
Init done
Expected value for L=8 is 7.1836656
4.4220 --------------------------------
4.4297 --------------------------------
4.4103 --------------------------------
Uliscan 21 Dec 98
L=8 256 258560
Measuring file moby-dick-ciph-aesca2-sbox-only.dat
Init done
Expected value for L=8 is 7.1836656
4.6556 -----------------------------------
4.6627 -----------------------------------
4.6468 -----------------------------------
Uliscan 21 Dec 98
L=8 256 258560
Measuring file moby-dick-ciph-aesca5-sbox-only.dat
Init done
Expected value for L=8 is 7.1836656
4.9300 -------------------------------------
4.9401 -------------------------------------
4.9197 -------------------------------------
Uliscan 21 Dec 98
L=8 256 258560
Measuring file moby-dick-ciph-aesca10-sbox-only.dat
Init done
Expected value for L=8 is 7.1836656
5.1294 ---------------------------------------
5.1403 ---------------------------------------
5.1277 ---------------------------------------
```

```
Uliscan 21 Dec 98
L=8 256 258560
Measuring file moby-dick-ciph-aesca50-sbox-only.dat
Init done
Expected value for L=8 is 7.1836656
5.3950 -----------------------------------------
5.4093 -----------------------------------------
5.3927 -----------------------------------------
```

C11. Conditional Entropy Results

*Removed all CBC code and padding code, and aligned input along 32 bytes.

```
X: magna-carta.dat
Y: magna-carta-fakeOTP.dat
Cond Entropy = 4.18532
EntropyX = 4.4147
EntropyY = 7.99284
X: magna-carta.dat
Y: magna-carta-ciph-AES.dat
Cond Entropy = 4.06934
EntropyX = 4.4147
EntropyY = 7.99436
X: magna-carta.dat
Y: magna-carta-ciph-AESCA1.dat
Cond Entropy = 4.12997
EntropyX = 4.4147
EntropyY = 7.99384
X: magna-carta.dat
Y: magna-carta-ciph-AESCA2.dat
Cond Entropy = 4.06829
EntropyX = 4.4147
EntropyY = 7.99345
X: magna-carta.dat
Y: magna-carta-ciph-AESCA3.dat
Cond Entropy = 4.10717
EntropyX = 4.4147
EntropyY = 7.99404
X: magna-carta.dat
Y: magna-carta-ciph-AESCA4.dat
Cond Entropy = 4.14187
EntropyX = 4.4147
EntropyY = 7.99505
X: magna-carta.dat
Y: magna-carta-ciph-AESCA5.dat
Cond Entropy = 4.07648
EntropyX = 4.4147
EntropyY = 7.99389
X: magna-carta.dat
Y: magna-carta-ciph-AESCA10.dat
Cond Entropy = 4.15784
EntropyX = 4.4147
EntropyY = 7.9943
X: moby-dick.dat
Y: moby-dick-ciph-fakeOTP.dat
Cond Entropy = 4.48107
```

```
EntropyX = 4.49714
EntropyY = 7.99972
X: moby-dick.dat
Y: moby-dick-ciph-AES.dat
Cond Entropy = 4.47943
EntropyX = 4.49714
EntropyY = 7.99973
X: moby-dick.dat
Y: moby-dick-ciph-AESCA1.dat
Cond Entropy = 4.46895
EntropyX = 4.49714
EntropyY = 7.99973
X: moby-dick.dat
Y: moby-dick-ciph-AESCA2.dat
Cond Entropy = 4.46718
EntropyX = 4.49714
EntropyY = 7.99973
X: moby-dick.dat
Y: moby-dick-ciph-AESCA3.dat
Cond Entropy = 4.47666
EntropyX = 4.49714
EntropyY = 7.99969
X: moby-dick.dat
Y: moby-dick-ciph-AESCA4.dat
Cond Entropy = 4.47919
EntropyX = 4.49714
EntropyY = 7.99974
X: moby-dick.dat
Y: moby-dick-ciph-AESCA5.dat
Cond Entropy = 4.47805
EntropyX = 4.49714
EntropyY = 7.99974
X: moby-dick.dat
Y: moby-dick-ciph-AESCA10.dat
Cond Entropy = 4.46592
EntropyX = 4.49714
EntropyY = 7.99972
```

C12. Compression Test Results

Uncompressed Results
```
-rw-r--r-- 1 shambler users  29824 2007-05-21 23:22 magna-carta-ciph-AES.dat
-rw-r--r-- 1 shambler users  29824 2007-05-21 23:22 magna-carta-ciph-AESCA1.dat
-rw-r--r-- 1 shambler users  29824 2007-05-21 23:23 magna-carta-ciph-AESCA10.dat
-rw-r--r-- 1 shambler users  29824 2007-05-21 23:23 magna-carta-ciph-AESCA2.dat
-rw-r--r-- 1 shambler users  29824 2007-05-21 23:23 magna-carta-ciph-AESCA3.dat
-rw-r--r-- 1 shambler users  29824 2007-05-21 23:23 magna-carta-ciph-AESCA4.dat
-rw-r--r-- 1 shambler users  29824 2007-05-21 23:23 magna-carta-ciph-AESCA5.dat
-rw-r--r-- 1 shambler users  29824 2007-05-22 00:13 magna-carta-ciph-fakeOTP.dat
-rw-r--r-- 1 shambler users  29824 2007-05-21 21:59 magna-carta.dat
-rw-r--r-- 1 shambler users 643232 2007-05-21 23:20 moby-dick-ciph-AES.dat
-rw-r--r-- 1 shambler users 643232 2007-05-21 23:21 moby-dick-ciph-AESCA1.dat
-rw-r--r-- 1 shambler users 643232 2007-05-21 23:24 moby-dick-ciph-AESCA10.dat
-rw-r--r-- 1 shambler users 643232 2007-05-21 23:21 moby-dick-ciph-AESCA2.dat
-rw-r--r-- 1 shambler users 643232 2007-05-21 23:21 moby-dick-ciph-AESCA3.dat
-rw-r--r-- 1 shambler users 643232 2007-05-21 23:21 moby-dick-ciph-AESCA4.dat
-rw-r--r-- 1 shambler users 643232 2007-05-21 23:21 moby-dick-ciph-AESCA5.dat
-rw-r--r-- 1 shambler users 643232 2007-05-22 00:51 moby-dick-ciph-fakeOTP.dat
-rw-r--r-- 1 shambler users 643232 2007-05-21 23:18 moby-dick.dat
```

Compressed Bzip2 Results

```
-rw-r--r-- 1 shambler users  30304 2007-05-21 23:22 magna-carta-ciph-AES.dat.bz2
-rw-r--r-- 1 shambler users  30302 2007-05-21 23:22 magna-carta-ciph-AESCA1.dat.bz2
-rw-r--r-- 1 shambler users  30303 2007-05-21 23:23 magna-carta-ciph-AESCA10.dat.bz2
-rw-r--r-- 1 shambler users  30317 2007-05-21 23:23 magna-carta-ciph-AESCA2.dat.bz2
-rw-r--r-- 1 shambler users  30304 2007-05-21 23:23 magna-carta-ciph-AESCA3.dat.bz2
-rw-r--r-- 1 shambler users  30316 2007-05-21 23:23 magna-carta-ciph-AESCA4.dat.bz2
-rw-r--r-- 1 shambler users  30317 2007-05-21 23:23 magna-carta-ciph-AESCA5.dat.bz2
-rw-r--r-- 1 shambler users  30319 2007-05-22 00:13 magna-carta-ciph-fakeOTP.dat.bz2
-rw-r--r-- 1 shambler users   9638 2007-05-21 21:59 magna-carta.dat.bz2
-rw-r--r-- 1 shambler users 646597 2007-05-21 23:20 moby-dick-ciph-AES.dat.bz2
-rw-r--r-- 1 shambler users 646299 2007-05-21 23:21 moby-dick-ciph-AESCA1.dat.bz2
-rw-r--r-- 1 shambler users 646375 2007-05-21 23:24 moby-dick-ciph-AESCA10.dat.bz2
-rw-r--r-- 1 shambler users 646345 2007-05-21 23:21 moby-dick-ciph-AESCA2.dat.bz2
-rw-r--r-- 1 shambler users 646698 2007-05-21 23:21 moby-dick-ciph-AESCA3.dat.bz2
-rw-r--r-- 1 shambler users 646678 2007-05-21 23:21 moby-dick-ciph-AESCA4.dat.bz2
-rw-r--r-- 1 shambler users 646576 2007-05-21 23:21 moby-dick-ciph-AESCA5.dat.bz2
-rw-r--r-- 1 shambler users 646571 2007-05-22 00:51 moby-dick-ciph-fakeOTP.dat.bz2
-rw-r--r-- 1 shambler users 200486 2007-05-21 23:18 moby-dick.dat.bz2
```

C13. Margolus Automata SDL Visualization Test

```cpp
#include <cstdlib>
#include <iostream>
#include <SDL/SDL.h>
#include <SDL/SDL_gfxprimitives.h>
#include <math.h>
#include <time.h>
#include <stdint.h>


#define HEIGHT 64
#define WIDTH 64
#define size 4


uint8_t thresh = 0x7f;


int swapCount=0;


uint8_t test[HEIGHT*WIDTH];
bool evenodd = true;

#define getCell(x,y) (test[fixX(x)+fixY(y)*WIDTH])
#define fixX(x) ((WIDTH + (x))%WIDTH)
#define fixY(y) ((HEIGHT + (y))%HEIGHT)

void swap(int x1,int y1,int x2,int y2)
{
    swapCount++;
    uint8_t temp = getCell(x1,y1);
    getCell(x1,y1) = getCell(x2,y2);
    getCell(x2,y2) = temp;
}

#define swapDiag1(x,y) swap(x,y,x+1,y+1)
#define swapDiag2(x,y) swap(x+1,y,x,y+1)
#define swapHoriz1(x,y) swap(x,y,x+1,y)
#define swapHoriz2(x,y) swap(x,y+1,x+1,y+1)
#define swapVert1(x,y) swap(x,y,x,y+1)
```

148

```cpp
#define swapVert2(x,y) swap(x+1,y,x+1,y+1)

enum{SCREENWIDTH = WIDTH*size,        SCREENHEIGHT = HEIGHT*size,
SCREENBPP = 32, SCREENFLAGS = SDL_HWSURFACE|SDL_DOUBLEBUF};
SDL_Surface* pSurface;
SDL_Event keyEvent;

using namespace std;

#define T true
#define F false

bool rules[16][6] =
{{F,F,F,F,F,F},{T,F,F,F,F,F},{F,T,F,F,F,F},{F,F,F,F,F,F},{F,T,F,F,F,F},
{F,F,F,F,F,F},{F,F,T,T,F,F},{T,F,F,F,F,F},{T,F,F,F,F,F},{F,F,T,T,F,F},{
F,F,F,F,F,F},{F,T,F,F,F,F},{F,F,F,F,F,F},{F,T,F,F,F,F},{T,F,F,F,F,F},{F
,F,F,F,F,F}};

void doTransition(int x, int y, uint8_t currState)
{
    for (int i=0;i<6;i++)
    {
        if (rules[currState][i])
        {
            switch (i)
            {
            case 0:
                swapDiag1(x, y);
                break;
            case 1:
                swapDiag2(x,y);
                break;
            case 2:
                swapHoriz1(x,y);
                break;
            case 3:
                swapHoriz2(x,y);
                break;
            case 4:
                swapVert1(x,y);
                break;
            case 5:
                swapVert2(x,y);
                break;
            }
        }
    }
}

void initMap()
{
    srand(time(0));
    for (int i=0;i<HEIGHT;i++)
    {
        for (int j=0;j<WIDTH;j++)
        {
```

```
                //test[i*WIDTH+j] =
128;//j/(float)WIDTH*256;//rand()%255;//rand()%127;
            if (i>= HEIGHT/4 && i < HEIGHT-HEIGHT/4 && j >= WIDTH/4 &&
j < WIDTH-WIDTH/4) test[i*WIDTH+j] = 128 + rand()%128;
        }
    }
    //test[1600] = 0xff;
    //test[221] = 0xff;
}


#define getConfiguration(i,j) ((getCell(i+1,j+1)>thresh?8:0) +
(getCell(i,j+1)>thresh?4:0) + (getCell(i+1,j)>thresh?2:0) +
(getCell(i,j)>thresh?1:0))

void nextGeneration()
{
    evenodd ^= 1;
    for (int i=evenodd;i<HEIGHT;i+=2)
    {
        for (int j=evenodd;j<WIDTH;j+=2)
        {
            doTransition(i, j, getConfiguration(i,j));
        }
    }
}

void randomThresh()
{
    thresh = rand()%255;
}

void draw()
{
    for (int i=0;i<HEIGHT;i++)
    {
        for (int j=0;j<WIDTH;j++)
        {
            uint8_t c = getCell(i,j);
            if (c > 127)
            {

boxRGBA(pSurface,i*size,j*size,(i+1)*size,(j+1)*size,255,255,255,255);
            }
            else
boxRGBA(pSurface,i*size,j*size,(i+1)*size,(j+1)*size,0,0,0,255);
        }
    }
}

void randomRules()
{
    srand(time(0));
    for (int i=0;i<16;i++)
        for (int j=0;j<6;j++)
        {
            rules[i][j] = 0;
```

150

```cpp
            rules[i][j] = rand()%50>25?T:F;
        }
}

int main(int argc, char *argv[])
{
    initMap();

    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        cout << "Error Initializing SDL Video" << endl;
        SDL_Quit();
        return 1;
    }
    pSurface = SDL_SetVideoMode ( SCREENWIDTH , SCREENHEIGHT ,SCREENBPP
, SCREENFLAGS ) ;

    while (1)
    {
        //nextGeneration();
        draw();
        SDL_Flip(pSurface);
        //SDL_Delay(40);

        SDL_PollEvent(&keyEvent);
        switch (keyEvent.type)
        {
        case SDL_KEYDOWN:
            switch (keyEvent.key.keysym.sym)
            {
            case SDLK_DOWN:
                SDL_Quit();
                return 0;
                break;
            case SDLK_UP:
                randomThresh();
                break;
            case SDLK_LEFT:
                //initMap();
                randomRules();
                break;
            case SDLK_RIGHT:
                SDL_SaveBMP(pSurface, "test.bmp");
                break;
            case SDLK_SPACE:
                //evenodd ^= 1;
                for(int i=0;i<5;i++)
                {
                    nextGeneration();
                    draw();
                    SDL_Flip(pSurface);
                }
                break;
            }
            break;
        }
        while (keyEvent.type == SDL_KEYDOWN){
```

```
            SDL_PollEvent(&keyEvent);
        }
    }

    system("PAUSE");
    SDL_Quit();
    return EXIT_SUCCESS;
}
```

7. DEVELOPMENT ENVIRONMENT

Slackware Linux 11.0:
     Bash 3.1.17
     GNU Nano Editor 1.3.12
     GCC 3.4.6
     GNU gprof 2.15.92.0.2
Windows XP SP2:
     BloodShed Dev-C++ 4.9.9.2
     Mingw/GCC 3.4.2
     SDL 1.2.8
     SDL_gfx 2.0.13
Other Software:
     Uliscan 21 Dec 98 (Mauer's)
     PuTTY 0.58
     WinSCP 4.0
     Microsoft Word
     Microsoft Excel
     Microsoft Visio
     Microsoft Paint
     PDF Creator