NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

SOLVING MULTI-VARIATE POLYNOMIAL EQUATIONS IN A FINITE FIELD

by

Natalie Vanatta

June 2013

Dissertation Supervisor: David R. Canright

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
30–5–2013	Dissertation	2009-02-01—2013-06-01

4. TITLE AND SUBTITLE

Solving Multi-variate Polynomial Equations in a Finite Field

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Natalie Vanatta

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Army

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

Solving large systems of multivariate polynomial equations is an active area of mathematical research, as these polynomials are used in many fields of science. The objective of this research is to advance the development of algebraic methods to attack the mathematical foundations of modern-day encryption methods, which can be modeled as a system of multivariate polynomial equations over a finite field. Our techniques overcome the limitations of previous methods. Additionally, a model is proposed to estimate the time required to solve large systems with our methods. All of these elements were tested successfully on AES and its predecessor, Square. The results showed our techniques to be comparable with a brute force technique. To the best of our knowledge, no other purely algebraic attack on AES has been shown to be this efficient.

15. SUBJECT TERMS

Finite Fields; AES; MRHS; multi-link; multi-agree; Square

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	281	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SOLVING MULTI-VARIATE POLYNOMIAL EQUATIONS IN A FINITE FIELD

Natalie Vanatta
Major, United States Army
B.E., Stevens Institute of Technology, 2001
M.S., Stevens Institute of Technology, 2001
M.S., Naval Postgraduate School, 2007
Submitted in partial fulfillment of the
requirements for the degree of
DOCTOR OF PHILOSOPHY IN APPLIED MATHEMATICS
from the
NAVAL POSTGRADUATE SCHOOL
June 2013

Author: Natalie Vanatta

David R. Canright
Associate Professor of
Applied Mathematics
Dissertation Advisor

Jon T. Butler
Distinguished Professor of
Electrical and Computer
Engineering

Håvard Raddum
Researcher
University of Bergen

Craig W. Rasmussen
Professor of Applied
Mathematics

Pantelimon Stănică
Professor of Applied
Mathematics

Approved by: Carlos F. Borges
Chair, Department of Applied Mathematics

Approved by: O. Douglas Moses
Vice Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Solving large systems of multivariate polynomial equations is an active area of mathematical research, as these polynomials are used in many fields of science. The objective of this research is to advance the development of algebraic methods to attack the mathematical foundations of modern-day encryption methods, which can be modeled as a system of multivariate polynomial equations over a finite field. Our techniques overcome the limitations of previous methods. Additionally, a model is proposed to estimate the time required to solve large systems with our methods. All of these elements were tested successfully on AES and its predecessor, Square. The results showed our techniques to be comparable with a brute force technique. To the best of our knowledge, no other purely algebraic attack on AES has been shown to be this efficient.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AES	Advanced Encryption Standard
DES	Data Encryption Standard
FIPS	Federal Information Processing Standards
MDS	Maximal Distance Separable
MQ	Multivariate Quadratic System of Equations
MRHS	Multiple Right Hand Sides
NIST	National Institute of Standards and Technology
NPS	Naval Postgraduate School
NSA	National Security Agency
SPN	Substitution Permutation Network
USG	United States Government
XL	eXtended Linearization
XOR	Exclusive OR (logical operation)
XSL	eXtended Sparse Linearization

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

This is my opportunity to offer a few words of thanks to those people who have been helpful, or at least extremely tolerant, during the gestation of this work. There is not enough space on the page to properly express gratitude for their enormous help, understanding, and support. I can only hope that each of you will realize how important you were, and you are, to me.

I am ever grateful to my advisor, Dr. Canright. I am thankful for his patience, guidance, and unwavering support during this research. Thanks also go to my committee members—Dr. Butler, Dr. Raddum, Dr. Rasmussen, and Dr. Stănică—for their guidance and support through this process.

I am deeply thankful for the support of my family and friends. To my parents—for their continuous support throughout my whole life. It is fair to say that without them, I would not have become who I am today. To some of my best friends in the world who are family to me—the Siltons and the Fletchers. Your loving support, unrelenting faith that I would eventually finish, and always available shoulders to lean on were priceless.

I am deeply appreciative to my Army mentors—COL Greg Conti, COL Charles Grindle, COL Charles Rimbey, and LTC Brian Lunday. Whether it was advice about this body of work or my career path or just life in general, you all have always made a difference and shown me the path to success. Thank you.

Finally, to my many readers of this dissertation—a heartfelt thanks. Whether you were a mathematician or a historian, you gave outstanding advice, and someday, I truly WILL learn how to avoid passive voice in my narration.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

The need to deliver secure information pervades society and daily life. Since Julius Caesar's time, leaders have attempted to protect important communications from being read by unauthorized people or governments. Caesar is credited with the first use of a cipher for military purposes when he sent a message to the besieged Cicero, who was on the verge of surrender. Caesar's best known cipher was one in which he replaced each letter by the letter three places down the alphabet. Thankfully, the design of cryptosystems in the modern age has significantly advanced from the days of the Roman Empire. Today, cryptosystems are systematic in nature; based on hard mathematical problems; and tested, probed and analyzed using significant computing power before being utilized. The result is a secure system which can both encrypt messages and decrypt the result.

The adjective "secure" in the crypto arena does not refer to absolute security but to computational security. The subtle difference is that modern cryptosystems are not unbreakable but are practically impossible to mount a practical attack against given today's computing power. The most obvious type of attack is a brute force attack. This occurs when an adversary attempts every possible key in the cryptosystem in order to determine which one was used to encrypt a certain message. Many modern cryptosystems have 2^{256} possible keys. To test all of these possibilities, all the known computing resources in the world would require more time than the remaining life of the universe. These systems are therefore considered impractical to break and are accepted as secure.

Within the cryptography community, the governmental designation of a cryptosystem as secure acts as a red cape to a bull—a challenge to attack. Unlike a bull, which seeks to gore the matador in a physical attack, analysts strive to find the key in a cryptologic attack.

Auguste Kerckhoffs, the 19th century Dutch linguist and cryptographer, wrote the defining work *La Cryptographie Militaire* in 1883. He had studied the most modern cryptosystems of the time and produced six enduring principles of cryptography. The most famous [4] is that "The security of a cryptosystem must not depend on keeping secret the crypto-algorithm. The security depends only on keeping secret the key." Therefore, we should assume that the enemy knows everything about the implementation of our cryptosystem, but as long as he does not have access

to the key, our communications are safe. Typically, attackers are assumed to have access to plaintext and its corresponding ciphertext generated by the algorithm's implementation (referred to as a plaintext-ciphertext attack). Therefore, an attacker's efforts are focused on deducing the only secret information—the key.

Over the years, a multitude of attack techniques (linear and differential cryptanalysis to name two recent ones) have been developed in an effort to break modern cryptosystems. The systems broken by these methods were largely abandoned in favor of ones more resistant to these known attacks. That begs the question, what is next? A portion of the cryptographic community [5] has turned to the concept of algebraic attacks. An algebraic attack focuses on the mathematical underpinnings of the encryption algorithm. It seeks to model the cryptosystem as a system of equations to be solved in order to obtain the key. This idea of transforming a cipher into a system of equations is not new. Claude Shannon [6] remarked on it in his seminal 1946 paper entitled *Communication Theory of Secrecy Systems* [7]: "solving a certain [crypto] system requires at least as much work as solving a system of simultaneous equations in a large number of unknowns, of a complex type." Claude Shannon is considered the father of the electronic communications age as he developed both the mathematical theory of communication and information theory in the 1930s and 1940s.

Given that Shannon's insight occurred in the 1940s, why have not these various algebraic attacks become commonplace? The answer is that these systems of equations are incredibly difficult to solve given the lack of sufficiently fast computers and efficient computer algebra techniques of the time. Now that we have significantly improved these resources, algebra has become a very active area of research within the field of cryptology in the last two decades.

1.1 Problem Background

Solving large systems of multivariate polynomial equations (as seen in Equation 1.1) is an active area of mathematical research. Polynomials are widely used in the sciences; for example they arise in robotics, coding theory, optimization, mathematical biology, computer vision, game theory, statistics, economics, physics, and many others. Unfortunately, there currently exists only a small suite of mathematical tools to find exact solutions for these systems. Additionally, many of these tools are limited in scope and require extensive resources (time and memory) to function. Current algebraic methods to solve these non-linear systems include Gröbner basis, linearization, F4, and F5. These general methods are discussed in Section 3.3.

$$A = \begin{cases} l_1(x_1, ...x_n) & = 0 \\ & \vdots \\ l_m(x_1, ...x_n) & = 0 \end{cases} \tag{1.1}$$

The many intriguing natural phenomenon and engineering marvels that are modeled with these equations explains why the search for a solution is so interesting to researchers. If the l_i consist of only quadratic, linear, and constant terms, Equation 1.1 is known as the MQ problem to mathematicians. MQ is shorthand for the problem of finding a solution to the Multivariate simultaneous system of Quadratic equations.

The MQ problem belongs to the set of NP-hard problems. The term Non-deterministic Polynomial-time Hard (NP Hard) refers to a complexity class of decision problems that are intrinsically harder than those that can be solved with a nondeterministic Turing machine in polynomial time. In simpler terms, it is a set of problems in which computers can not "find" a solution easily but can "check" the solution easily (i.e., in polynomial time). Since many people believe $P \neq NP$, it is widely thought that no polynomial time algorithm exists for the MQ problem.

In 2006 researchers Raddum and Semaev [8] created a new method to solve very specific systems of multivariate polynomial equations. Their method crafts multiple right-hand side equations (MRHS) to represent the system and then executes new processes to determine the solution. These processes are called linking, agreeing, gluing, guessing, and extracting. This method is described in Section 3.

This body of work details my journey of discovery and exploration into the world of algebraic cryptology. We extend their original ideas by crafting the MRHS system with field equations instead of bit equations. This improvement greatly reduces the number of variables and equations within the system. It also helps clarify the underlying structure of the system so that it can be used against itself. We also extend the ideas of linking and agreeing from pairs of equations (in Raddum's work) to arbitrary larger collections of equations. These new methods are described in Section 4. We have had success in applying these new tools to our specific system of multivariate polynomial equations, as seen in Section 5.

1.2 Overview

The remainder of the dissertation is organized as follows. Chapter 2 contains a brief overview of key algebraic, graph theoretic, and cryptologic concepts that the reader should be familiar with in order to best understand this body of work. Chapter 3 covers the creation, design methodology, and complete description of our target cryptosystem, the Advanced Encryption Standard (AES). Chapter 4 presents Raddum and Semaev's MRHS algorithm, examples of its key processes, and their computational results on attacking AES. Chapter 5 discusses our extensions of the original MRHS algorithm. Chapter 6 presents the experimental results from applying our method to AES. Chapter 7 showcases the new algorithm's application on other, similar block ciphers to AES with initial results. Chapter 8 covers future work and conclusions.

CHAPTER 2:
Background

2.1 Algebraic Background

In this section, some algebraic definitions and basics are discussed as they pertain to this research. For a more detailed treatment, consult an algebra book such as [9].

The group is the fundamental building block of algebra. Once a group is established, properties are added in order to create a ring, then an integral domain, then a field. This section traces that "creation" to a Galois Field with examples provided to assist the reader's understanding.

A ***group***, G, is a set of elements with a binary operation (called addition) such that for all a,b in R the following axioms hold:

1. Closure: If a and b belong to G, then $a+b$ is also in G.
2. Associativity: $a+(b+c)=(a+b)+c \quad \forall\, a,b,c$ in G.
3. Identity: $\exists e \in G$ such that $a+e=e+a=a \quad \forall\, a \in G$.
4. Inverse: $\forall\, a \in G$, $\exists b \in G$ such that $a+b=b+a=e$.

A group is designated as an abelian group if the addition operation is also commutative. Namely, that $a+b=b+a \quad \forall\, a,b$ in G.

A ***ring***, R, is a set of elements with two binary operations (called addition and multiplication) such that for all a,b,c in R the following axioms hold:

1. R is an abelian group with respect to addition.
2. Closure under multiplication: If a and b belong to R, then ab is also in R.
3. Multiplication is associative: $a(bc)=(ab)c \quad \forall\, a,b,c$ in R.
4. Left Distributive: $a(b+c)=ab+ac \quad \forall\, a,b,c$ in R.
5. Right Distributive: $(b+c)a=ba+ca \quad \forall\, a,b,c$ in R.

A ring is designated as a commutative ring if the multiplication operation is also commutative. Namely, that $ab=ba \quad \forall\, a,b$ in R.

An ***integral domain***, D, is a set of elements with two binary operations (called addition and multiplication) such that for all a,b,c in D the following axioms hold:

1. D is a commutative ring.
2. Multiplicative identity: $\exists$ an element $1 \in D$ such that $a1 = 1a = a \quad \forall\, a \in D$.
3. No zero divisors: If a, b in D and $ab = 0$, then either $a = 0$ or $b = 0$.

A ***field*** F is a set of elements with two binary operations (called addition and multiplication) such that for all a,b,c in F the following axioms hold:

1. F is an integral domain.
2. Multiplicative inverse: For each a in F, except 0, there is an element a^{-1} in F such that $aa^{-1} = (a^{-1})a = 1$.

A field, then, is a set of elements in which it is possible to do addition, subtraction, multiplication, and division without leaving the set. Subtraction is defined by: $a - b = a + (-b)$ using additive inverses. Division is defined by: $a/b = a(b^{-1})$ using multiplicative inverses.

A ***finite field*** is a field that has a finite set of elements. The number of elements in a field is called the order of the field. Any two finite fields of order q are isomorphic. ***Isomorphic*** means that the two fields have a one-to-one mapping between them that preserves the operations amongst the elements. There exists a finite field of order q if and only if $q = p^n$ where p is a prime number (called the characteristic of the field) and n is a positive integer. These fields are also referred to as Galois fields and written as GF(p^n).

When $n = 1$, the finite field is also referred to as a ***prime field***. Let p be a prime number. Then GF(p) is constructed as follows:

- The set of elements in this field are the integers modulo p: $Z_p = \{0, 1, ...p-1\}$.
- The first binary operation is addition modulo p. The additive identity is $e = 0$.
- The second binary operation is multiplication modulo p. The multiplicative identity is $e = 1$. Integers only have a multiplicative inverse in Z_n if they are relatively prime to n. Since p is prime, all positive integers less than p are relatively prime to p. Therefore, there exists a multiplicative inverse for all the non-zero elements in GF(p).

This research used only $p = 2$. In the case of GF(2), addition (and subtraction) is logical bit-wise XOR and multiplication is logical bit-wise AND.

Example in GF(2). GF(2) $= \{0, 1\}$.
$0 + a = a \qquad 1 + 1 = 0 \qquad 1 * a = a \qquad 0 * 0 = 0$

When $n > 1$, integers mod p^n cannot be the elements of the field as they do not all have multiplicative inverses within the field. Rather, GF(p^n) may be constructed as follows:

- The set of elements in this field are polynomials, in particular the set S of all polynomials of degree at most $n-1$ with coefficients in Z_p.
- The first binary operation is polynomial addition with coefficient arithmetic performed modulo p. The additive identity is 0.
- The second binary operation is polynomial multiplication which is performed modulo an irreducible polynomial $m(x)$ of degree n.

Each element of GF(p^n) has the form

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0 = \sum_{i=0}^{n-1} a_i x^i,$$

where $a_i \in \{0, 1, ..., p-1\}$ for each i $(1 \leq i \leq n-1)$. This definition will yield a total of p^n different polynomials.

Example of elements in GF(2^4). The elements of this finite field are the 16 binary polynomials of degree at most 3. The table below contains all elements of this field.

0	z^2	z^3	z^3+z^2
1	z^2+1	z^3+1	z^3+z^2+1
z	z^2+z	z^3+z	z^3+z^2+z
$z+1$	z^2+z+1	z^3+z+1	z^3+z^2+z+1

The first binary operation in GF(p^n) is polynomial addition, with coefficient arithmetic performed modulo p. If $f(x) = \sum_{i=0}^{k} a_i x^i$ and $g(x) = \sum_{i=0}^{m} b_i x^i$, with $k \geq m$ then addition is defined by $f(x) + g(x) = \sum_{i=0}^{m} (a_i + b_i)x^i + \sum_{i=m+1}^{k} a_i x^i$. Note that "+" is used to denote three types of addition: polynomial, field, and integer addition. Subtraction (inverse of addition) is done in a similar manner.

Example of addition in GF(2^4).

$$(z^3+z^2+1)+(z^2+z+1)=z^3+z$$

The second binary operation in GF(p^n) is polynomial multiplication which is performed modulo an irreducible polynomial $m(x)$ of degree n. This operation is defined by $f(x)\times g(x)=\sum_{i=0}^{k+m} c_i x^i$ where $c_j = a_0b_j + a_1b_{j-1} + ... + a_{j-1}b_1 + a_jb_0$. The degree of the polynomial answer is the sum of the degrees of $f(x)$ and $g(x)$. This is an issue if the resulting polynomial does not fall within the set of elements of the finite field. Therefore, a refinement is placed on the multiplication operation to ensure closure over the set. If a multiplication results in a polynomial of degree larger than $n-1$, then the polynomial is reduced modulo an irreducible polynomial $m(x)$ of degree n. An irreducible polynomial within a field is one that cannot be expressed as a product of two polynomials, both over Z_p, and both of degree lower than itself. In slang, this would be a prime polynomial for the field.

Example of multiplication in GF(2^4) with $m(x)=z^4+z+1$.

$$\begin{aligned}(z^3+z^2+1)\times(z^2+z+1) &= (z^5+z^4+z^3)+(z^4+z^3+z^2)+(z^2+z+1)\\ &= (z^5+z+1)\\ &= (z^5+z+1)\text{modulo}(z^4+z+1)\\ &= z^2+1. \end{aligned} \tag{2.1}$$

In this research, I use the Galois fields with $p=2$ and $n=2,4,8$ which are designated as GF(2^2), GF(2^4), and GF(2^8). Important features of these fields are the following:

- Galois fields of characteristic 2 can represent their elements as polynomials, binary or hexadecimal.
- Addition is performed as bit-wise XOR with an identity element of the polynomial 0 where each element is their own inverse.
- Multiplication has an identity element of the polynomial 1.

Example of arithmetic on different representations of the elements in GF(2^3)

$$\{57\}\oplus\{83\}=\{D4\}. \tag{2.2}$$

$$\{01010111\}\oplus\{10000011\}=\{11010100\}. \tag{2.3}$$

$$(x^6+x^4+x^2+x+1)+(x^7+x+1)=x^7+x^6+x^4+x^2. \tag{2.4}$$

Equation 2.2, 2.3, and 2.4 all hold the same values - their difference is in their representation of the elements of GF(2^8).

Example of different representations of the elements in GF(2^3)

polynomial	*binary*	*hexadecimal*
0	000	0
1	001	1
z	010	2
z^2	100	4
$z+1$	011	3
z^2+z	110	6
z^2+z+1	111	7
z^2+1	101	5

2.2 Graph Theoretic Background

In this section, some graph theoretic definitions and basics are discussed as they pertain to this research. For a more detailed treatment, consult a graph theory text such as [10].

A ***graph*** $G=\{V,E\}$ consists of a finite nonempty set V of objects called ***vertices*** and a set E of 2-element subsets of V called ***edges***. Vertices are also referred to as ***nodes***. When drawing a graph, the vertices are represented by points (or small circles) and edges are indicated by a line segment or curve between the two points in a plane.

In Figure 2.1, $V(G)=\{a,b,c,d,e,f\}$ and $E(G)=\{ab,bc,cd,bd,de,ef,df\}$. The number of vertices is the ***order*** of G. In order for the graph to exist, $|V(G)|>0$. The given graph has order 6. The number of edges is the ***size*** of the graph. The given graph has size 7.

Graphs are classified as directed or undirected. Figure 2.1 is an un-directed graph. A directed graph has edges that have orientation, which means that the edge set consists of ordered pairs of vertices. A ***simple*** directed graph allows only edges to occur between distinct vertices. An example of this is shown in Figure 2.2. An undirected graph has edges without orientation as in the graph in Figure 2.1. A complete graph is an undirected graph in which every pair of distinct

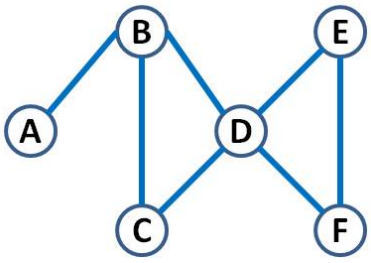

Figure 2.1: An undirected graph.

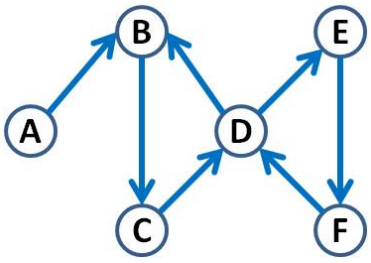

Figure 2.2: A directed graph.

vertices is connected by a unique edge. A subgraph is a graph, G', is a graph whose vertex set and edge sets are respectively subsets of the vertex and edge sets of G.

In this research, we also discuss bipartite graphs. A graph is bipartite if the V(G) can be partitioned into two subsets U and W such that every edge of G joins a vertex of U and a vertex of W. Figure 2.3 is an example of a bipartite graph with U(G)= $\{a\}$ and W(G)= $\{b,c,d\}$.

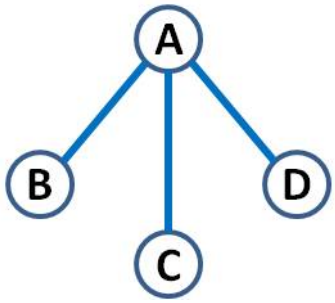

Figure 2.3: A bipartite graph.

More generally, a graph is a ***k-partite graph*** if V(G) can be partitioned into k subsets $V_1, V_2, ..., V_k$ such that uv is only an edge of G if u and v belong to different partite sets. If, in addition,

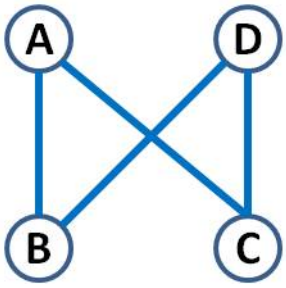

Figure 2.4: A complete bipartite graph.

every two vertices in different partite sets are joined by an edge, then G is a complete k-partite graph. [10] An example of a complete bipartite graph is in Figure 2.4.

A path is a sequence of edges which connect a sequence of vertices. A finite path has a start and end vertex. A closed path starts and ends at the same vertex. If a path does not visit any vertex more than once, it is a simple path. A path that is simple and closed is a cycle. A k-cycle is a cycle that visits k unique vertices.

2.3 Cryptographic Background

The fundamental objective of cryptology is to allow two individuals (normally referred to in literature as Alice and Bob) to communicate over an unsecure channel in such a way that a third person (Eve) cannot understand the dialog. Alice has a message (called plaintext) that she wants to pass to Bob. She encrypts the plaintext with a pre-designated key. The result is ciphertext which she transmits over the channel to Bob. While Eve is eavesdropping, she grabs a copy of the transmitted message but cannot decipher it to the plaintext because she does not know the key. Once Bob receives the ciphertext he can successfully re-create the original message because he knows the encryption key that Alice used [11].

A ***cryptosystem*** (or a cipher) is a five-tuple (P,C,K,E,D), where the following conditions are satisfied:

1. P is a finite set of possible plaintexts.
2. C is a finite set of possible ciphertexts.
3. K is the keyspace – the finite set of possible keys.
4. For each $k \in K$, there is an encryption algorithm $e_k \in E$ and a corresponding decryption algorithm $d_k \in D$. For $e_k : P \to C$ and $d_k : C \to P$ are functions such that $d_k(e_k(x)) = x$ for every plaintext $x \in P$.

The two most common types of cryptosystems are ***block*** and ***stream***. This refers to the amount of plaintext that the system encrypts at one time. It either operates on one bit (or byte) at a time or a group of bits (bytes) at a time.

A block cryptosystem can further be classified as symmetric or asymmetric. This classification refers to the type of key. A ***symmetric*** cryptosystem uses an encryption key that can be calculated from the decryption key. In fact, it might use the same key to encrypt and decrypt. This generates the requirement for Alice and Bob to agree on a key before secure communications

can begin. The Advanced Encryption Standard (AES) is the current standard for encryption in the United States and much of the world. It is discussed in Chapter 3. It is important to note that AES is a symmetric cryptosystem.

One specific symmetric block cryptosystem is the Feistel cipher. This is an iterated system with layers of diffusion and confusion in each round. A Feistel cipher block is split into two halves at the start of each round. Only one half of the block is transformed by the round's functions. At the end of the round, the block halves are swapped. Based on its structure, it is an easily reversible system allowing the same hardware to encrypt and decrypt messages. Many of the AES predecessors use this Feistel structure (i.e. DES, Lucifer, Blowfish). However, AES does not. Rather it uses the SPN (Substitution Permutation Network) scheme. In an SPN cipher, all bits are treated uniformly to both diffusion and confusion operations; each round consists of a substitution and a permutation. The result is a cipher that is efficient to implement in both hardware and software.

An ***asymmetric*** cryptosystem is also referred to as a public-key cryptosystem. It uses different keys for encryption and decryption. There is no known computationally feasible method to determine the decryption key from the encryption key. This feature enables complete strangers to use these systems.

Regardless of the classification of cryptosystem, the focus of cryptanalysts is to break the system. A cryptosystem is considered academically 'broken' when an attack can correctly determine the key quicker than a brute force attack. A brute force attack is an exhaustive key search, when an attacker systemically checks all possible keys until the correct key is found. Worse case, this could require the attacker to check every possible key within the key space. Although a cryptosystem may be theoretically broken, it could still be in use today if the attack is not practical with today's resources.

One of the most prevalent cryptosystems today is the Advanced Encryption Standard (AES) and therefore there is no larger, brighter red cape in the bull ring. This research will explore the continuation of Raddum and Semaev's idea for an algebraic attack on this cipher.

CHAPTER 3:
Advanced Encryption Standard (AES)

In January 1997, RSA Laboratories [12] issued a handful of cryptologic challenges to the public with significant cash prizes for anyone who could break the encrypted messages. One of these challenges was for a message encrypted by the Data Encryption Standard (DES), which was the standard for U.S. government encryption at the time.[1] Simultaneously, the National Institute for Standards and Technology (NIST) [13] began the process to find a replacement cipher for DES, a replacement that would be "an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive government information well into the next century."

In September of 1997, the NIST process began with an official request for proposals that was open to the international community. One of the main reasons behind conducting a public search for an algorithm was to allow NIST to leverage the best and brightest minds across a multitude of specialties, nationalities, and salaries. Ultimately, NIST believed that this global effort would yield high quality results and inspire public confidence in the security of the cipher as the public would no longer have to just solely rely on the National Security Agency (NSA)'s promise of security.

The main requirements for the replacement cipher (which was to be named AES) were that it: 1) be a block cipher, 2) support a block size of 128 bits, and 3) support key sizes of 128, 192, and 256 bits. Fifteen ciphers were submitted, including one cipher each from Australia, Belgium, Costa Rica, France, Germany, Japan, Korea, one from a mixed team (UK, Norway, Israel), two from Canadian teams, and five from American teams [14]. In August 1998, NIST hosted the first of three AES conferences in which the cryptology community gathered to review the submissions, listen to the authors, and begin full-frontal assaults on the worthiness of the submitted algorithms.

In August 1999, NIST picked the five finalists, of which three had authors from the United States, one had authors from Belgium, and one had a mix of international authors. The five finalists were tested by the USG, industry, and the global cryptology community for the next two years. These five algorithms were evaluated on several key categories: security, performance

[1] Responses to the DES Challenge: In 1998, the Electronic Frontier Foundation created the "DES Cracker" machine which found the key in 56 hours at the cost of $250,000 to build. In 1999, a distributed network was used via the Internet to brute force the DES key in 22 hours and 15 minutes.

(i.e., hardware, software, and smart cards), design features (i.e., simplicity), and implementation difficulty (i.e., flexibility). The final scores are in Table 3.1 with a higher number indicating a better evaluation. Ultimately none of these ciphers were found to be "bad" but Rijndael was "the most elegant of the final five candidates" [15]. Despite its beauty, most of the comments that NIST received from both industry and government agencies showed a dislike for Rijndael compared to its competitors due to its simplicity [16].

	Rijndael	Serpent	Twofish	MARS	RC6
General Security	2	3	3	3	2
Implementation Difficulty	3	3	2	1	1
Software Performance	3	1	1	2	2
Smart Card Performance	3	3	2	1	1
Hardware Performance	3	3	2	1	2
Design Features	2	1	3	2	1
TOTAL	16	14	13	10	9

Table 3.1: Official AES finalist results. After [3].

Yet, Rijndael scored higher than the competition for many reasons. First was its transparent, simple design that allowed for quick and accurate security estimates while clearly showing there were no hidden "back doors" for the government to exploit. Second, it was based on the byte, which made it more versatile than many of its competitors for implementation. Finally, its strong algebraic structure supported its ability to be represented (either all or in part) in different manners on different platforms.

Despite the criticisms, on 06 December 2001 the Federal Information Processing Standards (FIPS) 197 [1] was published which re-named the Rijndael entry to AES. It also decreed that U.S. sensitive but unclassified documents would use it. Rijndael was the Belgian entry from Joan Daemen and Vincent Rijmen. Unlike its predecessors, AES has proven resistant to linear and differential attacks to date.

3.1 AES Structure

Successful modern ciphers incorporate two important ideas that are credited to Claude Shannon: confusion and diffusion. ***Confusion*** exists when the relationship between the plaintext and the ciphertext is obscured. This relationship is dictated by the key. Therefore, the correlation between the statistics of the ciphertext and the value of the encryption key should be as complex as possible. ***Diffusion*** exists when the plaintext is dispersed across the breadth of the ciphertext.

Namely, one bit of the plaintext should affect many bits of the ciphertext in order to hide the statistical structure of the plaintext [17, p. 72-73]. AES utilizes both of these principles.

In AES, a block of plaintext is transformed into ciphertext via rounds of mathematical manipulation. The input to each round is a 128-bit block and the output is also a 128-bit block. The basic unit for AES is the byte, unlike its predecessors. This translates into 16 bytes of input and output for each round. The conventional view of AES places those 16 bytes into a 4x4 square array called the state array, as seen in Figure 3.1.

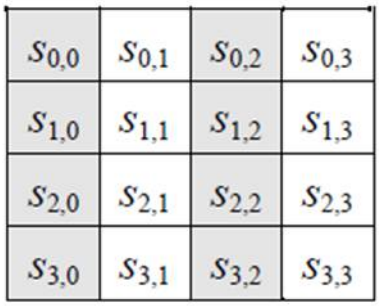

Figure 3.1: State array of AES. From [1].

3.1.1 Mathematical Background

A byte is an ordered sequence of eight bits $b_7b_6b_5b_4b_3b_2b_1b_0$ that can be thought of as a vector in eight dimensional vector space over a Galois Field of order 2. Since GF(2) has only two elements, and its arithmetic operations are simple in nature, much of the AES cryptanalysis work has focused on this representation. An alternative view would be to think of the byte as an element in GF(2^8). The specification for AES defines this finite field in terms of the following irreducible polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{3.1}$$

This field will be referred to as the ***Rijndael field*** for the remainder of this work. An AES byte (or word) can be referred to by its hexadecimal notation: {3A}, its binary representation: 00111010, or as an element in the Rijndael field: $x^5 + x^4 + x^3 + x$.

3.1.2 Encryption Algorithm

At the start of the AES algorithm, the plaintext is transferred into the state array and the 0th round key is added. Key addition is the first step in the algorithm because any transformation of the plaintext before the addition of a key does not contribute to the security of the cipher

as those steps can be easily stripped away during the cryptanalysis. Next, the round functions occur 10, 12, or 14 times. The resulting final State Array is the ciphertext.

The number of rounds in AES depends on the size of the key being used; a 128 bit key uses 10 rounds, a 192 bit key uses 12 rounds, and a 256 bit key uses 14 rounds. These systems will be respectively referred to as AES-128, AES-192, and AES-256. The number of required rounds was determined by identifying the maximum number of rounds for which shortcut attacks were found during the NIST competition and then adding a considerable safety margin [18, p. 41-42]. The inner workings of AES-128, AES-192, and AES-256 are the same. Within the rounds, four transformations occur that implement the diffusion and confusion principles. These four transformations of the state array are: 1) byte substitution using an S-box, 2) shifting rows, 3) mixing columns, and 4) adding a round key.

In each round, the Byte Substitution transformation occurs first. This is a simple substitution of one byte of the State Array for another possible byte using a statically defined substitution referred to as the S-box. This S-box contains a full permutation of the elements in the Rijndael field and is created by the composition of three operations. The first operation is a multiplicative inversion over the Rijndael field with the minor modification that $\{00\}$ is inverted to $\{00\}$. Therefore, the input byte x (as long as x$\neq$0) has an output of w such that $xw \equiv 1 \pmod{m(x)}$. The second operation is a GF(2) linear mapping. The output w from the inversion is regarded as a vector in $\mathrm{GF}(2)^8$. This is then multiplied by a specific circulant matrix to determine the output y. The third operation is the addition of a constant. The output y from the linear mapping is then added to the field element $\{63\}$ to produce the output of the S-box [19, p. 48].

The polynomial representation of Byte Substitution (the composition of the three described operations) is a very sparse polynomial:

$$05x^{254} + 09x^{253} + F9x^{251} + 25x^{247} + F4x^{239} + 01x^{223} + B5x^{191} + 8Fx^{127} + 63. \tag{3.2}$$

It is within this transformation (Byte Substitution) that AES receives its non-linearity. This is the core strength of the algorithm. According to Rijndael's creators, the inversion operation was selected to provide the non-linearity to the system because of its very simple algebraic structure especially when compared to the complicated non-linearity function used in DES. The reason behind the inclusion of the linear mapping was so that interpolation attacks would not be successful. This mapping has a very simple description but becomes quite complicated

when combined with inversion. Finally, adding {63} ensures that the S-box has no fixed points ($S(a) = a$) and no opposite fixed points ($S(a) = \bar{a}$). The specific combination of these three functions is what provides confusion in the cipher [18, p. 34-36].

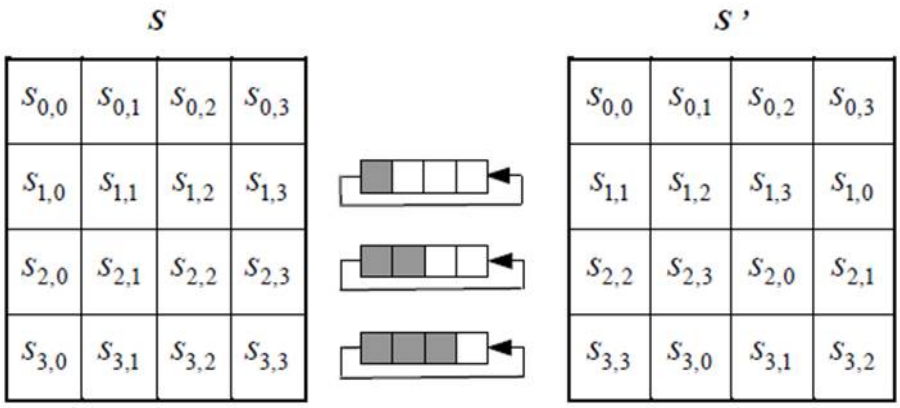

Figure 3.2: Shift Rows operation (s=initial array, s'=after array). From [1].

The Shift Rows operation is the second transformation in the round and is graphically depicted in Figure 3.2. Here the bytes in the *i*th row are rotated *i* places to the left. This operation gives high dispersion to the resulting state array.

The Mix Columns operation is the third transformation in the round and is graphically depicted in Figure 3.3. Here all the columns in the state array are independently mixed up. This is the result of multiplying each column by $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ and reducing modulo $x^4 + 1$. This can also be seen in Figure 3.3 as multiplying the current state array's columns by a specific MDS matrix (matrix where every square sub-matrix is invertible). The mixed column operation provides high local diffusion to the state array.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figure 3.3: Mix Columns operation on each column of the state array. From [1].

These two transformations (Shift Rows and Mix Columns) provide the algorithm its diffusion component. Both of these functions were specifically designed to adhere to the wide trail strategy which is a design approach for block ciphers that ensures they are resistant to linear and differential attacks while maintaining some semblance of efficiency. This strategy widens the probability trails for linear and differential attacks in order to make exploitation of the data more difficult [18, Ch. 9].

The fourth, and final, round transformation is to add the round key. The AES key schedule algorithm takes the initial key and creates a different 16-byte round key for each round. The Add Round Key function takes the round's unique key and adds it to the current state array. This completes a typical round of AES. These rounds are repeated 10, 12, or 14 times depending on the AES key size. The only modification to this process is that the final round of the algorithm does not use the Mix Columns operation. Therefore, the last round consists of a Byte Substitution, Shift Rows and Add Round Key. Here is how the typical process looks:

A SRMA . . . SRMA SRMA SRA.

The letters (in the above process) stand for the following transformations described in this section: A (Add Round Key), S (Byte Substitution), R (Shift Rows) and M (Mix Columns). As illustrated in this section, none of the operations used in AES are difficult. In fact, most of these values can be pre-computed and stored in look-up tables. This is the main reason why AES is a quick algorithm in both software and hardware implementations.

3.1.3 Key Schedule Algorithm

The key schedule algorithm creates the requisite number of round keys for the encryption algorithm. AES-128 requires 11 keys, AES-192 requires 13 keys, and AES-256 requires 15 keys. Similar to the encryption algorithm, the key creation process is almost the same for each of these systems; it just iterates longer to generate more key material for the larger systems. This research focuses on AES-128 so the key schedule algorithm presented here will be the variety for AES-128.

The algorithm begins with the user-provided key. It will use this key to recursively create the other round keys. The 128 bits of the key are placed in 4x4 matrix in a similar manner as the placement of the plaintext into the state array for the encryption process. However, the key schedule algorithm operates on 'words' which are 4-byte constructs as opposed to the

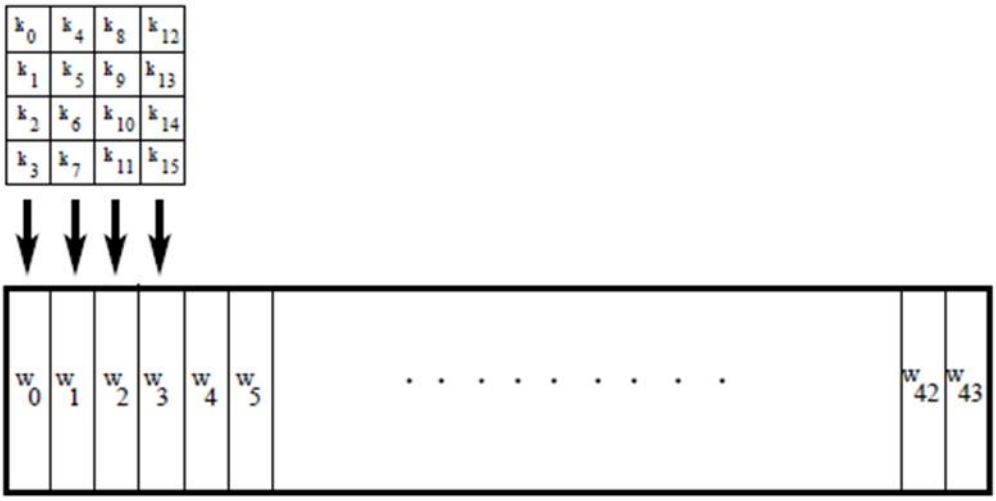

Figure 3.4: Start of the AES key schedule algorithm. From [2].

encryption algorithm which operates on bytes. These words are the columns of the 4x4 matrix. The algorithm begins when the four columns of the key matrix are loaded into the first four columns of the expanded key structure (see Figure 3.4). The first 4 bytes of the encryption key are word w_0, the second 4 bytes are word w_1, etc. The key schedule algorithm expands the first four words into a 44-word key schedule. Words 0 to 3 (the original key) are XOR'ed with the plaintext at the start of the algorithm. The remaining 40 words are used four at a time during each of the rounds. For instance, $w_4w_5w_6w_7$ is the Round 1 key, and so forth.

Figure 3.5 shows the flow of the key schedule algorithm. The algorithm works on four words at a time. Three of the four words are created as simple XORs with the previous round's key. For instance, $w_1 \oplus w_4 = w_5$ and $w_2 \oplus w_5 = w_6$ and $w_3 \oplus w_6 = w_7$. One of the four words is created using a more difficult process—it is transformed by the g function prior to XOR. For instance, $g(w_3) \oplus w_0 = w_4$. This process ensures that the previous round key influences the next round key [2].

The g function operates on the last word of the previous 4-word key. Let the bytes of this word be represented by $w_3 = [b_0b_1b_2b_3]$. The g function uses the following sub-functions:

1. RotWord performs a one-byte circular left shift on the word. Namely, $[b_0b_1b_2b_3]$ becomes $[b_1b_2b_3b_0]$.

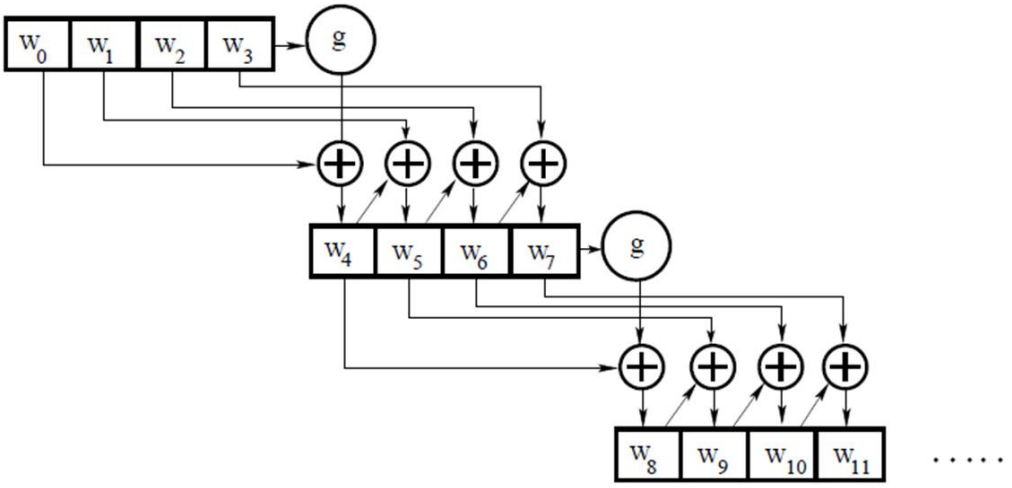

Figure 3.5: AES key schedule algorithm. From [2].

2. SubWord performs a byte substitution on each byte in the input word $[b_1 b_2 b_3 b_0]$ using the same S-box as the encryption algorithm.
3. The results are then XORed with a round constant.

The round constant is a word whose three right-most bytes are always zero. Therefore, for the ith round key, this constant is $(RC[i], 0, 0, 0)$. RC[i] is determined recursively:

$$\begin{aligned} RC[1] &= 1, \\ RC[j] &= 2 * RC[j-1]. \end{aligned} \tag{3.3}$$

The purpose of the round constant in the key schedule algorithm is similar to its use in the encryption algorithm—it removes the symmetries that might have been incurred by other steps of the process. This algorithm clearly uses diffusion as changing one bit of the user provided key will affect all of the round keys. After one round, there is a significant change in the key and by the last round, the key has changed in over half of the bits [17, p 173-174]. The key schedule algorithm is also efficient because it re-uses many of the functions from the encryption algorithm. It also introduces non-linearity (i.e., difficulty in solving) through the use of the proven AES S-box. Finally, unlike its predecessors, this algorithm ensures that AES has no weak keys.

3.1.4 Small Variants

In 2005, C. Cid et al. suggested creating small scale variants of AES to analyze [20]. The rationale for creating these small variants is that attempting to execute attacks on the full version of AES is infeasible given today's computing power. These versions retain, as much as possible, the algebraic properties of full AES. This allows researchers to test theories and attack methods on versions via current computing resources. AES is scaled in terms of four parameters:

1. number of rounds n $(1 \leq n \leq 10)$
2. number of rows in the State array r $(r = 1, 2, 4)$
3. number of columns in the State array c $(c = 1, 2, 4)$
4. the size of a word e $(e = 4, 8)$

The notation of a scaled version of AES is referred as *nrce*. Full AES is A448, written in hexadecimal notation.

These small scale variants of AES have a smaller key space than full AES so an exhaustive key search is feasible to attack them. Recall that their purpose is as a research tool for analysis and never for commercial implementation. Cid's original paper describes (in detail) the structure of the small scale variants to include the irreducible polynomials used for the various word sizes. It includes the S-box construction, linear mapping, and inversion look-up tables for these variants which have become the common framework that all researchers use in order to compare methods and analyze results with respect to AES.

For this research, I use all variants of Cid's small scale AES, and also variants with $e = 2$. The four element field (resulting from $e = 2$) enables me to perform the mathematical computations manually in order to test/validate the method and gain useful insights when e increases. However, $e = 2$ is not a good parameter for an actual cryptosystem since every possible S-box in the four-element field is affine. Therefore, none of them is non-linear, making the system of equations easier to solve.

3.2 Representation

The structure of AES allows it to be represented in different ways. Some of these different representations provide practitioners insights into how various portions of the algorithm interact with each other, some representations make the algorithm more resistant to side-channel attacks, and other representations might ultimately uncover exploitable weakness [19, p. 3].

Therefore, the simple algebraic structure of AES is where cryptanalysts are focusing their attacks. Recall that an algebraic attack is defined as an attempt to solve the multivariate polynomial equation system that represents the cipher in order to obtain the key. Simplistically, for each plaintext block pushed through AES, there exists a system of equations describing each round key and intermediate state arrays as the block traverses the algorithm.

The idea of representing a cipher as a system of equations should be a comfortable idea even to the mathematical layman. In middle school, most students learn that a mathematical function is akin to a black box in which one provides input, magic happens, and then output results as modeled in Figure 3.6.

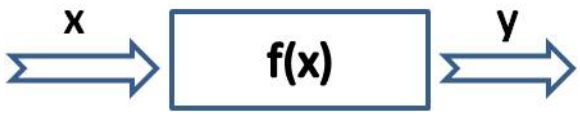

Figure 3.6: Generic representation of a function.

This middle school construct is merely a high-level view of what a cryptosystem is-—plaintext (x) is given to it and out comes ciphertext (y). However, leaving it at that level of detail creates a horribly complicated function. Ferguson, Shroeppel and Whiting [21] craft a single equation which consists of 2^{50} terms (with a leading term of x^{255}) for AES-128 in their paper. However, it is ultimately very impractical to solve. Instead of attempting to solve the cipher as one function, it can be subdivided into intermediate equations since all the components within the black box are known.

The resulting equation system has two parts. The first part is the set of equations that describe the encryption process. It is assumed that we know a plaintext / ciphertext pair generated by the system so that the first and final AES state arrays are known. This is due to Kerkhoff's principle. The unknowns in this system are the intermediate state arrays and the round keys. These equations are unique to the plaintext / ciphertext pair that influenced its creation. The second part of the system is the set of equations that describe the key schedule algorithm. These equations depict how the initial key is used to create individual round keys and hold true for the specific key used. The complete knowledge of the algebra in each round allows us to craft the system of equations that describe the state arrays and the key completely.

Some mathematicians choose to express the AES system of equations over GF(2)— in terms

of bits. This sparse system results in 14,976 equations over 4,288 variables [19, p. 78]. By using linear substitutions to remove variables, a more compact quadratic system of equations is created. This system has 9,600 equations over 1,600 variables [19, p. 79]. Other mathematicians create the system over GF(2^8)—in terms of bytes. While there are many different ways to describe the system of equations representing AES, the primary goal should be to use a representation that helps one attack or solve it.

Given that we now think of a cryptosystem as just a system of equations, the idea of breaking the system is analogous to solving the system for the key variables. Yet, recall that the strength of AES is found in the inversion done in the Byte Substitution function at the beginning of each round. This causes any system of equations defining AES to be non-linear, which turns solving the system into a hard problem.

As mentioned in Chapter 1, the problem of solving a multivariate quadratic (MQ) system of equations is a known NP-hard problem. However, Shamir et al. [22] demonstrated that the complexity of the MQ problem drops substantially when the system is over-defined. Therefore, many cryptanalysts have focused on the GF(2) representation of AES to take advantage of the fact that the equation system is both over-defined and sparse.

3.3 Solving Methods

The important question is whether any of these representations can reduce the complexity of the problem or have a structure that can be manipulated to find a solution in practical time. Three main tools exist to solve large systems of multivariate polynomial equations representing a cryptosystem. They are linearization, Gröbner Bases, and Boolean propagation.

3.3.1 Linearization

The first method uses the idea of repeatedly linearizing the system of equations. The idea was first introduced by A. Kipnis and A. Shamir [23]. In general, we do not know how to solve the MQ problem. However, we do know (albeit, not efficiently often times) how to solve a system of linear equations. Therefore, we want to transform the multivariate quadratic equations into linear equations. Assume there exists a system of polynomial equations with m variables: $x_0, x_1, \ldots, x_{m-1}$. For every product $x_i x_j$, create a new term y_{ij} to replace it. Therefore, each time a quadratic term is replaced, more equations can be constructed and the process repeats until all that remains is a system of linear equations. Unfortunately, this method only solves the system of equations if there are at least $\frac{m^2}{2}$ equations for the m variables. If there are fewer polynomial

equations, then the resulting system of linear equations is under-defined and therefore, cannot be solved.

The downfall of this method is that a significant number of new variables are introduced through this process. Therefore, great care needs to be taken in its implementation. The two best known linearization algorithms are XL and XSL. When the eXtended Linearization (XL) attack was published, its authors hoped it would run in subexponential time but that seems unlikely based on current analysis. To exploit the sparseness of the system and resolve the inefficiencies of XL, the eXtended Sparse Linearization (XSL) algorithm was created. XSL is a tailored attack on symmetric block ciphers that use S-boxes, linear diffusion, and XOR their round key (like AES, Shark, Square and Serpent) [24].

3.3.2 Gröbner Bases

The second method to solve large systems of multivariate polynomial equations is to use the ideal derived from the system. Namely, transform the given system of equations into another set of polynomial equations with certain "nice" properties referred to as the Gröbner Basis. These systems are equivalent because they generate the same ideal. The classical technique for calculating Gröbner bases is the Buchberger algorithm. The Buchberger algorithm first sets the monomial order and then by computing the S-polynomial of two equations, eliminates the top monomial. This process repeats until all of the Gröbner bases are found and all but one of the variables are eliminated. This transforms the system into an univariate polynomial equation [5, pg 245-248].

The downfall of the second method is that the degree of the remaining monomials increase rapidly so the algorithm's time complexity makes this method impractical to use if there are many variables in the original system. Worse case, the algorithm's runtime is double exponential time ($f(x) = a^{b^x}$). The F4 algorithm and the F5 algorithm are two variants of this method that are used today. In practice, these algorithms cannot handle systems with more than 15 variables.

The F4 algorithm [25] is a strategy for executing the steps of the Buchberger algorithm which takes advantage of fast linear algebra techniques and the sparseness of the polynomials. The two advantages of this technique are that: 1) the memory requirements are controlled due to the pre-processing phase and 2) it reduces the system to row echelon form in order to use Gaussian elimination on the sparseness. F4's main problem lies with its tendency to produce

false positive solutions. Therefore, F5 was designed by J.C. Faugere [26] to combat this issue. The F5 algorithm is optimized and uses different criteria to deal with unnecessary critical pairs, but there is no complete proof that it will always find an answer.

3.3.3 Boolean Propagation

The third method to solve large systems of multivariate polynomial equations is based on the Boolean satisfiability (SAT) problem. The process begins by expressing the system as a complicated Boolean expression involving a number of variables. This expression is true if and only if the ciphertext equals the encryption of the plaintext. The process then consists of a search for key bits that make the expression true. This is accomplished by assigning values to variables until a conflict is reached (i.e., the Boolean expression becomes false). Then the process back-tracks and re-assigns values to remove the conflict. The key is revealed once the set of values is found that makes the entire expression true. Some classic SAT solvers [27] are: zChaff, MiniSat, WalkSAT and SAT4J. There are two classes of algorithms to solve this problem: 1) conflict-driven clause-learning algorithms (like Chaff) and 2) stochastic local search algorithms (like WalkSAT).

All of these methods have been tried on AES with limited success. As of today, the most successful published attack on AES is by Bogdanov et al. [28] in 2011. Compared to brute-force methods, this technique provides an advantage of about a factor of three to five. However, it requires 2^{88} pieces of data to solve the system for the key. The results are in Table 3.2.

AES key size	Computational Complexity
128	$2^{126.1}$
192	$2^{189.7}$
256	$2^{254.4}$

Table 3.2: Results of best attack on AES.

Therefore, AES has been broken. But Bogdanov's approach is clearly not practical with today's computing resources. Therefore, this research looks at a non-typical representation of the system of equations in order to develop a new method to solve them. This representation is discussed in Chapters 4 and 5.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
MRHS

This chapter summarizes the approach and methods of Raddum and Semaev. It begins with a discussion about how they represent the target system as a system of Multiple Right Hand Sides (MRHS) linear equations. The MRHS representation concept has been used in other mathematical systems, but using it on AES is the original work of Raddum and Semaev [8]. This representation is quite compact, and computations performed upon it can be more efficient compared to Chapter 3's algorithms. The chapter then discusses their methods (agreeing, gluing, guessing, and extracting) to solve the system.

4.1 Representation

Let $x = (x_1 x_2 \dots x_n)^T$ be a column vector of n Boolean variables, A be a $k \times n$ binary matrix of full rank, and $b_1, b_2, b_3, \dots, b_s$ be column-vectors of size k. An equation

$$Ax \in \{b_1, b_2, ..., b_s\} \tag{4.1}$$

is called an MRHS system of linear equations with right hand sides $b_1, b_2, b_3, \dots, b_s$. A solution to (Equation 4.1) is a Boolean n-vector satisfying one of the particular linear equations $Ax = b_i$. The set of all solutions to (Equation 4.1) is the union of solutions to the individual linear systems for all b_i. In order to more easily illustrate the possibilities in the MRHS, the $\overrightarrow{b}$ are written next to each other and the result called a matrix $\{L\}$. Therefore, (Equation 4.1) can be re-written as $Ax \in \{L\}$. The braces around the L serve as a reminder that this is not a traditional matrix, but instead an enumeration of the possible right-hand sides as a set of columns. Raddum and Semaev routinely called these systems symbols, notated within this body of work as: $S_1 : A_1 x \in \{L_1\}$. Our notation change (from the original notation $S_1 : A_1 x = [L_1]$) is intended to make the meaning more clear.

Therefore, AES is translated into a system of symbols:

$$S_1 : A_1 x \in \{L_1\}, \ \dots \ , S_m : A_m x \in \{L_m\}. \tag{4.2}$$

A solution to (Equation 4.2) is the $\overrightarrow{x}$ such that each of the symbols is satisfied (all of the underlying linear equations are satisfied). Suppose that there exists a unique solution $\overrightarrow{x}$ to a set of symbols, denoted x_0. Then $A_i x_0$ will be equivalent to only one column in $\{L_i\}$, which is the only possible right hand side. The selection of any other column in $\{L_i\}$ will result in an inconsistency in another symbol and therefore the system (Equation 4.2) will not be solved.

Raddum and Semaev originally created four algorithms that are used to solve the system of equations generated in Equation 4.2. They are known as agreeing, gluing, extracting and guessing. All four processes seek to remove the inconsistent columns, b_i, from each symbol.

4.2 Agreeing

Agreeing, in the context of Raddum and Semaev's work, is performed pair-wise between symbols. Let $S_i : A_i x \in \{L_i\}$ and $S_j : A_j x \in \{L_j\}$ be any two symbols within the system of equations. If S_i and S_j agree for every $b \in L_i$, then there exists a $b' \in L_j$ such that

$$\begin{pmatrix} A_i \\ A_j \end{pmatrix} \overrightarrow{x} = \begin{pmatrix} b \\ b' \end{pmatrix} \tag{4.3}$$

is consistent and vice versa. If S_i and S_j do not agree, then one removes the b columns from L_i in which $A_i x = b$ is inconsistent with all $A_j x \in \{L_j\}$. Similarly, one removes the b' columns from L_j in which $A_j x = b'$ is inconsistent with all $A_i x \in \{L_i\}$.

Raddum and Semaev created an algorithm for agreeing that utilizes typical linear algebra techniques. Let $A = \begin{pmatrix} A_i \\ A_j \end{pmatrix}$ be the concatenation of the matrices in A_i and A_j defined above. Matrix A has $t = k_i + k_j$ rows. Let $T_i = \begin{pmatrix} L_i \\ 0 \end{pmatrix}$ and $T_j = \begin{pmatrix} 0 \\ L_j \end{pmatrix}$ be matrices with t rows.

$$Ax = [T_i] + [T_j]. \tag{4.4}$$

In Equation 4.4, we can now pick one column from T_i and one from T_j, add them, and that is a possible right hand side for the new symbol. If A has full rank, then the two symbols agree.

4.2.1 Agreeing Procedure

1. Produce a non-singular transform matrix $U = U_{ij}$ of size $t * t$ such that UA is a matrix with all zeros in the last $r = r_{ij}$ rows and of rank $t - r$.
   (a) If $r = 0$, then symbols agree.
   (b) If $r > 0$, there are linear dependencies among the rows of A. Proceed to step 2.
2. Compute UT_i and UT_j. Let P_i denote the set of UT_i column projections in the last r coordinates. Similarly, define P_j.
   (a) If all the columns in P_i exist in P_j, then the symbols agree.
   (b) If $P_i \neq P_j$ then right hand sides can be removed. Proceed to Step 3.
3. For any column in P_i that is not found in P_j remove the corresponding columns in L_i. Likewise, for any columns in P_j that is not in P_i remove the corresponding column in L_j. The symbols now pair-wise agree.

The agreeing algorithm is the heart of Raddum and Semaev's solving process. Provided that the conditions can be set in order to reach step 3, a solution is quickly reached. Unfortunately, the AES system of symbols begins in an agreed state. It takes multiple iterations of the other algorithms (gluing, guessing, extracting) to create a situation where $r > 0$. In general, agreeing does not make a significant contribution to the process until right-hand sides are removed. However, then it is very effective.

The interesting aspect about the agreeing algorithm is that it can gain momentum to solve the system once RHSs are removed. For instance, without loss of generality, suppose the symbols S_h and S_i agree but S_i and S_j disagree. Therefore, columns could be removed from L_i or L_j. If columns are removed from L_i, perhaps now S_h and S_i no longer agree. Therefore, S_h and S_i will have to be re-agreed, and perhaps columns from L_h can now be removed. This can create a 'cascade effect' on the system and the system quickly reduces its size and complexity.

Agreeing Example [8]
Let there be two symbols: $A_1X \in \{L_1\}$ and $A_2X \in \{L_2\}$ in variables $X = \{x_1, x_2, x_3, x_4, x_5\}$ expressed in Equation 4.5.

$$\begin{pmatrix}11000\\10100\\10010\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\\x_4\\x_5\end{pmatrix}=\begin{bmatrix}1001\\0100\\0011\end{bmatrix},\quad \begin{pmatrix}01001\\00101\\00011\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\\x_4\\x_5\end{pmatrix}=\begin{bmatrix}0100\\1100\\1101\end{bmatrix} \tag{4.5}$$

These symbols can also be written in algebraic normal form as the following

$$x_1x_4+x_1x_2+x_2x_4+x_2+x_3+x_4+1=0,\quad x_2x_3+x_2x_5+x_3x_4+x_4x_5+x_2+x_3=0\,.$$

Step 1 of the agreeing process: Matrix A is produced and transformed with the matrix U:

$$A=\begin{pmatrix}11000\\10100\\10010\\01001\\00101\\00011\end{pmatrix}\rightarrow UA=\begin{pmatrix}11000\\10100\\10010\\01001\\00000\\00000\end{pmatrix},U=\begin{pmatrix}100000\\010000\\001000\\000100\\110110\\101101\end{pmatrix}. \tag{4.6}$$

Clearly, $r=2$ and A lacks full rank. Proceed to step 2 and create the following:

$$T_{12}=\begin{pmatrix}1001\\0100\\0011\\0000\\0000\\0000\end{pmatrix}\quad\text{and}\quad T_{21}=\begin{pmatrix}0000\\0000\\0000\\0100\\1100\\1101\end{pmatrix}.$$

This enables UT_{12} and UT_{21} to be computed.

$$UT_{12}=\begin{pmatrix}1001\\0100\\0011\\0000\\1101\\1010\end{pmatrix}\quad\text{and}\quad UT_{21}=\begin{pmatrix}0000\\0000\\0000\\0100\\1000\\1001\end{pmatrix}. \tag{4.7}$$

Examine the last two coordinates of the columns in Equations 4.7 and determine the projections:
$Pr_{12} = \{(1,1),(1,0),(0,1)\}, \quad Pr_{21} = \{(1,1),(0,0),(0,1)\}, \quad Pr_{12} \cap Pr_{21} = \{(1,1),(0,1)\}$
Notice that the second and fourth columns of UT_{12} do not match any columns of UT_{21}. That is, the projections show that the sum of the columns cannot give zero in the last two rows. Therefore, the second and fourth columns of L_1 should be removed. Similarly, the second and third columns of L_2 should be removed. The new symbols are:

$$\begin{pmatrix} 11000 \\ 10100 \\ 10010 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{bmatrix} 10 \\ 00 \\ 01 \end{bmatrix}, \quad \begin{pmatrix} 01001 \\ 00101 \\ 00011 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{bmatrix} 00 \\ 10 \\ 11 \end{bmatrix}, \tag{4.8}$$

and they now agree.

4.3 Gluing

Once all the symbols are pair-wise agreed and if the cryptosystem is not solved (i.e., all symbols reduced to one RHS), then Raddum and Semaev utilize a gluing algorithm to create new symbols. Suppose you glue symbols S_i and S_j into a new symbol $Bx \in \{L\}$ for which the set of solutions is the common solutions to $A_1x \in \{L_1\}$ and $A_2x \in \{L_2\}$. Once the new symbol is created, S_i and S_j are removed from the system because their relevant information is now captured in the new symbol.

The gluing procedure is straightforward to execute. B is the submatrix of UA (using previous example) in its first $t-r$ nonzero rows. The matrix L has $t-r$ rows and its columns are created by adding one column from UT_i to one column of UT_j if they have the same projection in the last r-coordinates. L contains the first $t-r$ entries of the newly created column.

Gluing Example [8]
Using the symbols resulting from agreeing in Equation 4.8, the two symbols are glued together into one symbol. The result is shown in Equation 4.9.

$$\begin{pmatrix} 11000 \\ 10100 \\ 10010 \\ 01001 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{bmatrix} 10 \\ 00 \\ 01 \\ 00 \end{bmatrix}. \tag{4.9}$$

Although gluing is required to eliminate RHSs in Raddum and Semaev's process and therefore allow agreeing to cascade, it comes at a great cost. In fact, the practicality of the entire process is rendered ineffective by the use of this algorithm because the storage requirements for these new, significantly larger symbols grows exponentially as the process continues. If L_i and L_j have width s_i and s_j respectively, their glued symbol could have as many as $s_i \times s_j$ columns. Implementations of this algorithm have a built-in threshold to restrict gluing operations based on the available computer system resources. After gluing a pair of symbols, the new symbol will normally not be in agreement with all the other symbols in the system, so the agreeing algorithm must usually be re-run on the system in hope of eliminating more columns in the $[L]$s.

In Raddum and Semaev's work, they never specify how to choose which symbols to glue. Work has been done on crafting different selection methods however this process is still hampered by its extensive memory requirements.

4.4 Extracting

The third component of Raddum and Semaev's MRHS solving process consists of extracting the linear equations once they are found in the modified system. Recall that the end state of this process is a solved system of equations which result in the key variables used in the cipher. If the multivariate system is reduced to a linear system, then this is trivial to solve with any modern computer algebra program. As the agreeing and gluing procedures remove right-hand sides, they can also leave behind linear equations when enough RHSs are removed in a symbol. For instance, choose a symbol $S : Ax \in \{L\}$ where L is a $k \times s$ matrix. Find a non-singular transformation matrix V of size $k \times k$ where VL is upper-triangular with zeros in the last r rows. Let A' be the submatrix of VA in the last r rows. Then $A'x = 0$ is the system of all independent linear homogeneous equations satisfied by the solutions of $Ax \in \{L\}$. Of course, non-homogeneous equations can also be extracted.

Once extracted, these linear equations help reduce the remaining symbols in the system.

Extracting Example [8]
Begin with Equation 4.9 from the gluing example. This symbol is then transformed with linear algebra row transformations. The symbol becomes Equation 4.10.

$$\begin{pmatrix} 11000 \\ 01010 \\ 10100 \\ 01001 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{bmatrix} 10 \\ 11 \\ 00 \\ 00 \end{bmatrix}. \tag{4.10}$$

This gives three linear equations: $x_2 + x_4 = 1, \quad x_1 + x_3 = 0, \quad$ and $\quad x_2 + x_5 = 0$. These three linear equations are equivalent to the system of two initial quadratic equations.

4.5 Guessing

Theoretically, the application of the three previous algorithms should be sufficient to solve the system. However, due to the thresholds placed on gluing, the system remains unsolved. Therefore, guessing became a component of the Raddum and Semaev's method. Once the entire system is pair-wise agreed, glued to its maximum threshold, and all linear equations have been extracted (and it is still not solved), then the only option is to start guessing some of the variables. Most of the implementations of the MRHS idea are performed on GF(2). Therefore, when a variable is guessed, either $x_i = 0$ or $x_i = 1$ is added as a linear equation. Several guesses can produce a system of linear equations that is a new symbol, S_0. This symbol can now be used in the agreeing and gluing processes. If a guess was wrong, it will manifest itself within the system by either forcing all the right hand sides of a symbol to be deleted or by creating a system of linear equations in S_0 that are inconsistent. Since each variable has only two possible values, if a guess was wrong, it is trivial to determine the correct value for a variable.

Although Raddum and Semaev never provide any specific techniques or ideas on how to efficiently guess variables, some work has been done in this area. Regardless of the method, with a threshold of 2^{16} for the maximum size of a symbol as a result of gluing operations, AES-128 still requires 112 of the 128 bits of the key variable to be found through guessing in order to solve the system.

4.6 Experimental Results

The results that Raddum and Semaev achieve are based on guessing variables in order to find the solution. Thresholds (θ) were emplaced on gluing. Table 4.1 gives the number of bits that have to be guessed in a key in order to find the correct solution to the system of equations.

	$\theta = 2^8$		$\theta = 2^{16}$			
	$n11e$	$n21e$	$n22e$	$n24e$	$n42e$	$n44e$
n	8-bit key	16-bit key	32-bit key	64-bit key	64-bit key	128-bit key
3	0	5	16	48	48	112
4	0	8	16	48	48	112
5	0	8	16	48	48	112
6	0	8	16	48	48	112
7	0	8	16	48	48	112
8	0	8	16	48	48	112
9	0	8	16	48	48	112
10	0	8	16	48	48	112

Table 4.1: Number of guesses to solve AES for various thresholds (θ).

CHAPTER 5:
Extending MRHS Methods

Raddum and Semaev's idea (as described in Chapter 4), while technically sound, is not currently a practical tool to break AES with today's computing power. This research takes a new look at their ideas in order to exploit/analyze AES's underlying structure in an effort to create a solving process that is more feasible. We adapt the Raddum and Semaev MRHS representation to the byte field rather than the bit field. In this field, addition is still bitwise XOR, and we used log and antilog tables for field multiplication. We apply our methods to AES and related block ciphers that utilize operations on the field of bytes. The use of bytes greatly simplifies the representation of AES variants as MRHS systems.

Specifically, it is the elegance of Raddum and Semaev's pair-wise agreeing method that is the foundation to their MRHS solving process. Therefore, we first look at how this method works visually on the data set. This leads to insights on the structure and behavior of the data that enabled a new, more successful concept of agreeing.

5.1 Representation

Recall the order of the transformations on the state arrays in AES (pg. 18)

$$\text{A} \quad \text{SRMA} \ldots \text{SRMA} \quad \text{SRMA} \quad \text{SRA} .$$

Mathematically speaking, SR is equivalent to RS—both orderings result in the same intermediate state arrays. In one, the S-box substitution occurs first and then the results are shifted within the row. In the other, the bytes are shifted and then the substitution occurs. The two operations are commutative with respect to each other. Therefore, an equivalent representation of the AES encryption process is:

$$\text{ARS} \quad \text{MARS} \ldots \text{MARS} \quad \text{MARS} \quad \text{A}. \tag{5.1}$$

The difficulty in solving AES's system of equations is the non-linear portion resulting from the inversion in the S-box. With the ordering shown in Equation 5.1, the S-box is the last operation in each segment. The steps to the left of the S denotes the input into the S-box. The output would then affect the next segment. These mathematical operations are conducted within a

finite field. Therefore, there are only a limited number of values that the S-box can produce, giving us the ability to enumerate them.

Let x be the column vector of the system unknowns. This would include the key variables and variables representing each intermediate state array. For purposes of this equivalent representation, an intermediate state array is the contents of the AES state array between each segment in Equation 5.1. Therefore, one can determine a linear combination of these variables that represent the input to a certain segment's S-box. Likewise, the output of the S-box can be also determined as a linear combination of the variables. This is possible because all of the AES operations (outside of the S-box) are linear in nature. All of these linear steps are incorporated into the linear side A.

Then each segment of the AES equivalent can be represented as a 2 row equation (Raddum and Semaev called them symbols, page 27) as follows:

$$\begin{bmatrix} \text{input to S box} \\ \text{output of S box} \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} \text{all possible input values} \\ \text{corresponding output} \end{bmatrix}. \tag{5.2}$$

In Equation 5.2, the rows of matrix A represent the input and output of the S-box (respectively). Similarly, the two-row vectors b represent all the possible field elements (as input to the S-box) and their corresponding S-box output. We know that at least one of these columns is correct for the system. Prior to the start of this research, we assumed that there would be a unique solution to the system. However, this is not always the case for smaller versions of AES. This is because a cryptosystem only requires that, under a specific key, a plaintext to ciphertext mapping is 1-1 not that only one key can yield a specific plaintext to ciphertext mapping. See Results (Chapter 6) for further discussion on this topic.

Once we find the correct input and resulting output value in Equation 5.2, then the equation set is linear and easily solved.

5.2 Notation

The following notation is used to create the equations: n is the number of rounds, P is plaintext, C is ciphertext, I is the intermediate state array after a round key is added, and X is the state array at the end of the byte substitution operation. Then the process looks like (with output

[states])

$$[P]\,A\,[I_0]\,RS\,[X_1] \quad MA\,[I_1]\,RS\,[X_2] \quad \ldots \quad MA\,[I_{n-2}]\,RS\,[X_{n-1}] \quad MA\,[I_{n-1}]\,RS\,[X_n] \quad A\,[C].$$

This process yields the following algebraic equations:

$$I_0 = P + K_0 \quad X_i = S(R(I_{i-1})) \quad I_i = M(X_i) + K_i \quad C = X_n + K_n \tag{5.3}$$

These translate into the following MRHS equations:

$$\begin{gathered}
\begin{bmatrix} K_{0,R(j,k),k} \\ X_{1,j,k} \end{bmatrix} = \begin{bmatrix} S_{\text{in}} \\ S_{\text{out}} \end{bmatrix} + \begin{bmatrix} P_{R(j,k),k} \\ 0 \end{bmatrix} \\
\begin{bmatrix} I_{i-1,R(j,k),k} \\ X_{i,j,k} \end{bmatrix} = \begin{bmatrix} S_{\text{in}} \\ S_{\text{out}} \end{bmatrix} \text{ where } \quad 2 \le i \le n-1 \\
\begin{bmatrix} I_{n-1,R(j,k),k} \\ K_{n,j,k} \end{bmatrix} = \begin{bmatrix} S_{\text{in}} \\ S_{\text{out}} \end{bmatrix} + \begin{bmatrix} 0 \\ C_{j,k} \end{bmatrix}
\end{gathered} \tag{5.4}$$

Note that R is just a reordering that moves words to other columns in the same row, so, in terms of indices

$$R(j,k) = (j+k) \mod c\,,$$

where (i,j,k) notation refers to (round, column, row) of the appropriate state array and c is the number of columns in the state array. The expressions S_{in} and S_{out} are the collection of inputs and outputs from the S box. The key schedule equations will be written in terms of columns:

$$K_{i,0} = F(K_{i-1,c-1}) + (\text{if } c > 1)\; K_{i-1,0} \quad K_{i,j} = K_{i-1,j} + K_{i,j-1} \quad i,j > 0, \tag{5.5}$$

where row k of the output of F is

$$F_k(K_{i,j}) = S(K_{i,j,(k+1 \mod r)}) + 2^i \text{ (if } k = 0)\,.$$

This gives the MRHS equation of

$$\begin{bmatrix} K_{i-1,c-1,(k+1 \mod r)} \\ K_{i,0,k} \end{bmatrix} + \begin{bmatrix} 0 \\ K_{i-1,0,k} \end{bmatrix} (\text{if } c > 1) = \begin{bmatrix} S_{\text{in}} \\ S_{\text{out}} \end{bmatrix} + \begin{bmatrix} 0 \\ 2^i \end{bmatrix} (\text{if } k = 0) . \quad (5.6)$$

Equations 5.4 and 5.6 fully define AES in MRHS notation. See Appendix C for the C code that creates the AES MRHS equations.

Example of MRHS Equations for 2424 There are 24 equations for the AES case of 2424. 8 MRHS equations are from the key schedule algorithm and 16 are from the AES encryption algorithm. There are 24 variables in the case of 2424. The first 16 are key variables (8 represent the initial key and 8 represent the first column of the other round keys). 8 variables are from the intermediate state arrays.

These first 8 MRHS equations are from the key schedule algorithm.

$$\begin{bmatrix} 000001000000000000000000 \\ 100000001000000000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDEF \\ 7A453F6B8CED2019 \end{bmatrix}$$

$$\begin{bmatrix} 000000100000000000000000 \\ 010000000100000000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDEF \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 000000010000000000000000 \\ 001000000010000000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDEF \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 000010000000000000000000 \\ 000100000001000000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDEF \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 000001000100000000000000 \\ 000000001000100000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDEF \\ 49760C58BFDE132A \end{bmatrix}$$

$$\begin{bmatrix} 000000100010000000000000 \\ 000000000100010000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDEF \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 000000010001000000000000 \\ 000000000010001000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDEF \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 00001000100000000000000 \\ 00000000001000100000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDEF \\ 6B542E7A9DFC3108 \end{bmatrix}$$

These 16 MRHS equations are from the AES encryption algorithm. Note that these equations have the Mix Columns operations and therefore, have 2 and 3 in matrix A.

$$\begin{bmatrix} 10000000000000000000000 \\ 00000000000000010000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 67452301EFCDAB89 \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 00000100000000000000000 \\ 00000000000000001000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 32107654BA98FEDC \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 00100000000000000000000 \\ 00000000000000000100000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} AB89EFCD23016745 \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 00000001000000000000000 \\ 00000000000000000010000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 45670123CDEF89AB \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 00001000000000000000000 \\ 00000000000000000001000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 1032547698BADCFE \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 01000000000000000000000 \\ 00000000000000000000100 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 67452301EFCDAB89 \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 00000010000000000000000 \\ 00000000000000000000010 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 76543210FEDCBA98 \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 00010000000000000000000 \\ 00000000000000000000001 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 89ABCDEF01234567 \\ 6B542E7A9DFC3108 \end{bmatrix}$$

$$\begin{bmatrix} 00000000100000002311000 \\ 00000000000010000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDE \\ E3DCA6F21574B980 \end{bmatrix}$$

$$\begin{bmatrix} 00000100010000000000001231 \\ 00000000000010000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDE \\ 94ABD1856203CEF7 \end{bmatrix}$$

$$\begin{bmatrix} 00000000010000011230000 \\ 00000000000001000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDE \\ F2CDB7E30465A891 \end{bmatrix}$$

$$\begin{bmatrix} 00000010001000000003112 \\ 00000000000000100000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDE \\ 0D32481CFB9A576E \end{bmatrix}$$

$$\begin{bmatrix} 00001000100000000002311 \\ 00001000100010000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDE \\ 85BAC0947312DFE6 \end{bmatrix}$$

$$\begin{bmatrix} 00000000100000012310000 \\ 00000100010010000000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDE \\ D0EF95C126478AB3 \end{bmatrix}$$

$$\begin{bmatrix} 00000010001000000001123 \\ 00000010001000100000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDE \\ 49760C58BFDE132A \end{bmatrix}$$

$$\begin{bmatrix} 00000000000100003112000 \\ 00000010001000100000000 \end{bmatrix} \overrightarrow{x} = \begin{bmatrix} 0123456789ABCDE \\ 85BAC0947312DFE6 \end{bmatrix}$$

The plaintext that created this example is 66A81374. The associated key was 635241A9. The ciphertext that resulted from AES was 8F96EB2E.

Theorem 1. *For this representation of AES variants, we always get k equations in k variables.*

Proof. This is a combinatorial proof. First, we count the variables and then count the equations. There are two types of variables: key variables and intermediate state array variables. Some of the key variables come from the initial key and others come from the subsequent round keys. The initial key has $r \cdot c$ words (a.k.a. variables). The subsequent round keys are defined using Byte Substitution on one column of the key state array. The remainder of the round key is generated by linear operations. Therefore, there are $n \cdot r$ key variables for the rounds. The state arrays have $r \cdot c$ words. The initial state array is the plaintext which is known. The last state array is the ciphertext which is known. That leaves $n-1$ intermediate state arrays that are unknown. This makes the total number of state array variables as $(n-1)(r \cdot c)$. Let k be the total number

of variables. Then $k = rc + nr + (n-1)rc = nrc + nr = nr(c+1)$. Now, count the number of MRHS equations. There are two types of equations: key schedule equations and encryption equations. The key schedule equations express the first column of the key matrix for each round. There are r entries in the first column. Therefore, there are $n \cdot r$ key schedule equations. Each round of the encryption algorithm has an S-box. The encryption equations model the input and output of the S-box. Therefore, there are $n \cdot r \cdot c$ equations to describe each block of the state array for each round. The total number of MRHS equations are $nr + nrc = nr(c+1) = k$. Therefore, our representation of AES variants will always have k equations in k variables. □

Theorem 2. *The coefficient matrices are independent of field size.*

Proof. When creating the equations, A's rows are linear combinations of the variables representing the linear components of the AES algorithm. Of the linear operations, it is only during the Mix Column transformation that a variable would be multiplied by a number other than 0 or 1. Recall that the Add Round Key function adds one value to another value; therefore, the entries in the coefficient matrix A are either 0s or 1s. The Shift Row function moves the value from one location to another; therefore, the entries in the coefficient matrix A are either 0s or 1s. The Mix Column function effectively multiples the current state array by a matrix with 0, 1, 2, and 3 as coefficients. All of the entries in the Mix Column matrix (see Figure 3.3) were 01, 02, and 03. Each of these numbers requires only 2 bits to represent. Therefore, whether the field size is 4, 16, or 256 is irrelevant. All numbers used in the AES MRHS equations are contained in the last two bits so the values of the co-efficient matrix A are the same. Therefore, the equations do not change based on the use of GF(2^2), GF(2^4), or GF(2^8). □

Once the MRHS equations are constructed for a certain variant of AES, the next step is to solve by removing RHSs until only one RHS exists per symbol. The resulting $\overrightarrow{x}$ has the key variables within it and yields a set of linear equations which can be trivially solved.

5.3 Graphical Representation

By design, AES has a clear structure that can be expressed mathematically or algorithmically. Graph theory can also describe this structure. It is upon the visual exploration of AES's structure that new insights can be gained and current solving procedures refined.

The AES system of MRHS symbols can be represented by a graph. Let a node represent a RHS in a symbol. If a RHS from symbol A and a RHS from symbol B agree (namely, they are consistent in their solution($\overrightarrow{x}$)), then there exists an undirected edge between them. By Raddum and Semaev's definition of pairwise agreement, these two symbols agree if every RHS (node) in symbol A is connected to a RHS (node) in symbol B, and vice versa. Figure 5.1 illustrates two symbols in GF(2^2) agreeing. This is a bipartite graph.

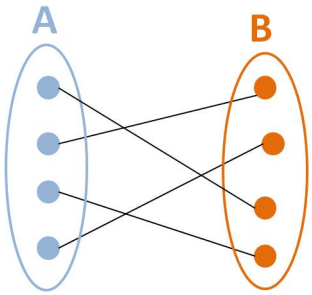

Figure 5.1: AES; RHS agreement.

As Raddum and Semaev's agreeing algorithm continues on all pairs of symbols, the result will be a k-partite graph where k is the number of symbols. The resulting graph depicts information about the relationships between the symbols. Eventually, this information will yield which RHSs are inconsistent. This will enable their representing nodes to be eliminated from the graph. Recall that an edge signifies agreement. Mathematically, this translates into the existence of a $\overrightarrow{x}$ that keeps both symbols consistent (i.e., solvable). The fact that one $\overrightarrow{x}$ works on all symbols is what classifies it as a solution to the system. Therefore, the final solution would be represented as a complete graph which uses one node from each MRHS symbol, as seen in Figure 5.2.

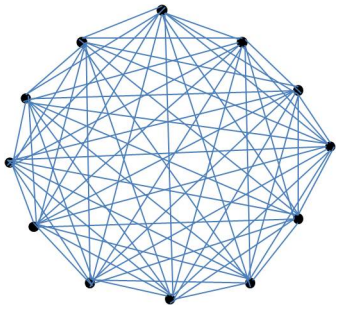

Figure 5.2: Graphical representation of the solution of a system with 12 symbols.

5.3.1 Agreement

Visually, Raddum and Semaev's pairwise-agreeing concept is equivalent to determining whether a connected, bipartite graph exists between every two symbols. If a node has no neighbors within the connected graph, then it can be discarded. This procedure is executed on all pairs of symbols. The final solution is represented as a complete graph on n nodes (see Figure 5.2). To be the solution means that there exists a consistent $\overrightarrow{x}$ amongst all symbols. Additionally, by definition of a complete graph, any subset of the final n nodes is also complete. These observations led to the idea of applying Raddum and Semaev's pair-wise agreeing algorithm to three symbols at a time. Under this idea, for any three symbols, if a RHS is not included in the connected 3-partite graph—then it can be discarded. Additionally, if a RHS is not used in a 3-cycle, then it can also be discarded.

This is repeated for all combinations of three symbols. By now having two criteria for RHS removal, we thought RHSs could be deleted faster in the process. A 3-cycle is more complex than a 2-cycle (yet still easy to detect on a graph using a depth first search algorithm). Therefore, the assumption is that they should be less common within a graph. This should result in elimination of nodes more quickly than Raddum's original process.

The initial implementation of the 3-agree concept used the information already captured in the original pairwise agreeing procedure. Namely, the implementation constructed an adjacency matrix representation of symbol A + symbol B agreeing, symbol B + symbol C agreeing, and symbol A + symbol C agreeing. Search algorithms were then used to search the matrix for any RHS nodes that were not used in a 3-cycle. The visualization in Figure 5.3 was what we hoped to find, in which there is only one 3-cycle and, therefore, it represents the solution. While the concept was sound in theory, the AES equations did not have any RHSs that were not in a 3-cycle. Namely, for any combination of three symbols, there exists a complete graph on the three symbols. More importantly, the sets of the three RHSs were unique. This result is shown in Figure 5.4. Therefore, given n RHS in a symbol, there were n complete graphs (representing possible solutions) to the three symbols.

The concept of 3-agree, while not the panacea, does yield information about the relationships between various edges and nodes. During testing, it highlights that if a RHS could be removed from any symbol, then a "cascade" of removals occurs from the graph which ultimately result in the solution. Unfortunately, the relationships do not tell us which of the RHSs to remove to achieve the correct solution. Since information is discovered about the AES equation system

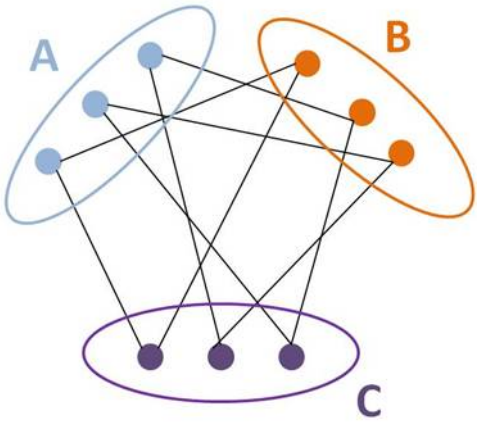

Figure 5.3: What we hoped to find in 3-agree process.

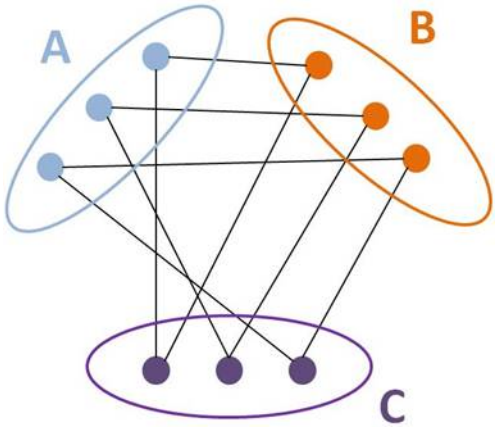

Figure 5.4: What we found in 3-agree process.

from both the 2-agree and 3-agree algorithms, this leads to the idea of scaling the process to four symbols and larger. However, the increase in the number of symbols to agree leads to a significant increase in the size of the adjacency matrix to store. Assuming that there are r RHS for each symbol and we want to n-agree the system; there are $(nr)^2$ elements within the matrix and r of the n-cycles to check. This brings us back to the original Raddum and Semaev problem: excessive memory requirements preclude finding a solution.

However, there is something to this idea. If a different concept of multi-agree (that is more memory-friendly) could be found—perhaps it could delete RHSs sooner. This would result in a solution without the need to guess a single variable.

Problem: Create a multi-agree algorithm that is memory-friendly. It should successfully capture all the relevant calculations/relationships between symbols but does not need to store the information. It will be enough to see the behavior that the symbols have with each other in order to determine what action (action = removal of RHS) causes the quickest cascade to the solution.

5.4 Links

What are the required conditions to remove a RHS from a symbol? Recall Raddum and Semaev's procedure (for pairs of symbols) to create the UA matrix (page 29). In order to be able to remove RHS from a symbol, the UA matrix (resulting from two symbols) must have $r > 0$. This condition is equivalent to the existence of a linear combination of rows of A that equal 0. Our new method (which extends Raddum and Semaev's concept) works on arbitrary sets of equations (what Raddum and Semaev called symbols) to include using all of the equations at once.

Unlike the crafted example in Chapter 4, in reality AES symbols typically yield $r = 0$ for most pairs of symbols. Therefore, if we could systematically find these linear combinations considering all the rows of all symbols, then this is called ***multi-linking***. In addition, examining all the rows of all symbols at the same time is quicker than individually checking all possible smaller combinations of symbols. These linear combinations (considering all the available information) generate multi-links.

Therefore, a group of symbols is termed linked if their resulting UA has $r > 0$. This means that their concatenation of A matrices is not of full rank, and therefore can be agreed. The number of symbols used in a link can range from 2 to n, with n referring to the total number of symbols in the system.

5.4.1 Where do links come from?

It is the structure of the AES system's equations that dictates the genesis of the links. The spareness of the AES structure creates sparse links. These links connect the AES variables through the various rounds of AES. The concatenation of the matrices A from the equations in the link have linearly dependent rows. Therefore, they must share some subset of variables. Hence, links relate variables to each other through the various stages of the encryption algorithm.

As the AES system is not a randomly generated system but comes from a specific algorithm, the

structure of the equations is predictable based on *nrce* parameters. Therefore, there is benefit to examining how the system and its links change by modifying the *nrce* parameters. Perhaps the information gleaned from smaller cases of AES will apply to the full version as the same type of equations are similarly linked over various iterations of *nrce*. This is the idea that lessons learned from attacking smaller variants of AES will yield relevant information about attacking the full version of AES.

5.4.2 How are links created?

Our links (which we refer to as multi-links to acknowledge their different creation technique compared to Raddum and Semaev's links) result from the application of linear algebra techniques on our MRHS equations. We begin by stacking all the A matrices together from the individual MRHS equations. Then, we row reduce the resulting large matrix using a LU decomposition technique. The resulting L matrix contains the record of the elementary row operations that produced the U (upper triangular) result. It is the bottom portion of L, corresponding to the all-zero rows of the reduced U matrix, that is the matrix of our multi-links. Therefore, our multi-links represent the dependence relations of A. We think of our multi-links as the basis for our link space, the left null space of the big A matrix. As a basis, this independent set of links captures all of the information about the linear part of the MRHS system.

Our computer algorithm quickly finds a set of multi-links. We refer to this set as the default set of links. It is interesting to note that many of our links have a preponderance of key schedule equations within them. This should be helpful as the key variables (which are used in the key schedule equations) are the most important variables in the system. With more relationship information about them, the system should be easier to solve for them.

Linear algebra tells us that any linear combination of a basis for a vector space is also an element of the vector space. Therefore, any linear combination of our multi-links will provide an alternate link in the link space. Solving the MRHS system with the new multi-links will also yield the solution. We refer to these alternate multi-links in this research as dependent links. This is in order to differentiate them from our default set of generated multi-links. Our computer program has the added functionality to create these user suggested dependent links. It creates a custom designed multi-link that features specific equations which should help solve the target system. Once the user has provided all the custom links that they need, then the algorithm will create the remainder of the multi-links necessary to fully describe the link space. This is akin to using a different basis to describe the system. These custom multi-links can occasionally be

helpful to solve the system when using more than one plaintext/ciphertext pair. See Section 6.4 for a discussion on using more than one pair of plaintext/ciphertext.

Theorem 3. *For our representation of AES MRHS equations, there are (at least) k multi-links associated with them.*

Proof. The AES MRHS equations have k variables in k equations. Take all the A matrices of the MRHS equations and stack them. The result is a matrix with $2k$ rows and k columns. Any matrix can be reduced by elementary row operations to a matrix in reduced row echelon form. A matrix in reduced row echelon form can have a maximum of one pivot per row. Therefore, the stacked A matrix has, at most, k pivots. All rows without a pivot are rows of zeros. Therefore, there are at least k rows of all zeros because there are a total of $2k$ rows in A. The rows of all zeros result from a distinct linear combination of the original rows that equal 0. This is the definition of our multi-links. Therefore, we will always have at least k multi-links for the system. Also, since the L matrix is row-equivalent to an identity matrix, these multi-links are linearly independent. □

Theorem 4. *The link structure is independent of e.*

Proof. The left side of the MRHS equations is the same no matter what field size used in this research. Therefore, the row reduction of the A matrices is the same no matter the field size. The multi-links come from the elementary row operations conducted on the A matrices. Therefore, the multi-links are the same for each field size used in this research. □

5.5 Multi-Agreeing

Raddum and Semaev's method of agreeing used their links (consisting of only two symbols) and was performed on one link at a time. The issue is that AES begins in an agreed state. This forced them to use the techniques of gluing and guessing to find a solution to the system. These are expensive operations (in both time and memory). We now have our new multi-links. We could try to agree each multi-link one at a time (like Raddum and Semaev) but we run into the same problem, that AES still starts in an agreed state. Therefore, we extend Raddum and

Semaev's agreeing concept in a similar fashion to what we did with the links. Instead of just looking at one multi-link at a time, we will look at several multi-links at a time. While there could exist more elegant ways of conducting this operation, our method is straightforward and successful.

Our new process is called multi-agreeing to differentiate from Raddum and Semaev's process. It is based on the idea that the generated set of multi-links can be agreed in a specific order to yield the answer (i.e., solve the system). Agreeing the multi-links in any order will solve the system, as they contain all the relationship information about the system. However, some orderings of the multi-links return the solution more efficiently than others. First, I will explain the concept of multi-agreeing and then discuss the methodology for creating an efficient ordering.

Start with a multi-link; call it L_i. This multi-link consists of the information from n equations and utilizes m variables. Operating within a finite field, there are a limited number of possible values for each variable. In fact, since each equation has 2^e different RHSs to choose from, then there are 2^{en} possible combinations of RHSs. However, the multi-link specifies a linear dependency amongst the equations. Recall the corresponding linear combination of RHSs must sum to the value zero, in order for that combination of RHSs to be consistent with that multi-link. We now use this information to choose one RHS of the combination, given the other $n-1$ RHSs of the combination. So, there are $2^{e(n-1)}$ combinations of RHSs of the equations involved in the multi-link that are consistent with the multi-link. These possibilities can be visually represented as complete subgraphs on the graph, and we refer to them as RHS sets.

Then we proceed to add one multi-link at a time to the current set of 'agreed' multi-links. Choose another link L_j which includes s equations that are not in L_i or any other previous link. The RHSs of these new equations are then constrained only by the new linear dependence relation of the new multi-link: one new relation involving s new choices. So RHSs can be freely chosen for $s-1$ of these new equations, and we can solve for the RHS of the remaining new equation. Hence

$$\text{the new number of RHS sets} = (\text{previous total}) \cdot (2^e)^{s-1}. \tag{5.7}$$

Based on Equation 5.7, each new link that is agreed in this process can increase, decrease, or not change the number of consistent RHS sets. Namely, the number of consistent RHS sets is increased whenever two or more new equations are added to the current listing of agreed

links via a new link. The number of consistent RHS sets remains the same when only one new equation is added to the set. The number of consistent RHS sets is decreased when a new link uses no new equations but adds additional relationship information into the agreed list of links instead. This case results in the number of possible solutions being reduced by a factor of the field size and possibly RHSs being eliminated. We call this case a "check-for-consistency" amongst all the information already gathered about the variables and their relationship to each other.

As more and more links are agreed, it becomes quite infeasible to store (in memory) the values of the consistent RHS sets. In our multi-agreeing process, it is enough to know the number of sets that exist at any given point. Recall that the intent of the multi-agreeing process is to eliminate RHSs. Therefore, until we reach a point where that is possible—it does not matter what the specific consistent RHS sets are.

It is only in the third case (number of consistent RHSs decrease) that we actually need to check all the RHS sets to see if a RHS can be deleted. Since we don't keep track of the various RHS sets, we must recreate them each time that we hit a check-for-consistency. The total number of consistent RHS sets can grow quite large, so we do not store them in memory. Instead, the process just tracks if a specific RHS can be deleted (i.e., it does not appear in any consistent set of RHSs). This methodology allows us to turn Raddum's memory-intensive process into a time-intensive process. Therefore, we want to keep the possible solutions sets small (to speed up the process) which means we want to keep the number of new equations (also known as the number degrees of freedom in the creation of consistent RHS sets) added each time to be small.

Towards the end of the multi-agreeing process, RHSs are eliminated from equations due to the checks for consistency. This is because eventually the system contains enough information about the relationships between the variables that some of the edges in the graph can be removed and, ultimately, nodes (RHSs) are detached from the connected graph. This means that they do not contribute to the final solution (recall Figure 5.2), and they can be deleted from our MRHS equations.

In this way, we reach the solution to the AES system of equations by only using the processes of multi-linking and multi-agreeing. Both of these processes require relatively little memory and can run on a standard PC. For the small-scale AES variants in this research, I used a 2.20 GHz AMD processor with 4 GB of total memory. Table 5.1 shows the time required to solve a sample of AES cases. The AES variants of 6442 and 7442 were not computed in this research

as they were not necessary to find the order for A442. It is our belief that the code used to implement multi-agreeing could be modified to take advantage of multiple computer processors when searching for RHSs to be deleted. This would significantly speed up the times recorded in this dissertation.

nrce	time	*nrce*	time	*nrce*	time
3424	0h 50m	3244	0h 13m	3442	0h 36m
4424	1h 26m	4244	1h 28m	4442	1h 59m
5424	2h 32m	5244	1h 49m	5442	3h 12m
6424	3h 46m	6244	2h 50m		
7424	5h 12m	7244	3h 40m		
A424	5h 36m	A244	7h 12m	A442	17h 29m

Table 5.1: Sample of AES solve times.

5.5.1 How decide order for multi-agreeing?

Given n links there are $n!$ possible orders to agree the generated links. Order matters because the number of new equations brought into the agreed set each time determines how often and how many RHS sets are checked within the program. Therefore, we strive to make the agreement order the most efficient in terms of size of sets.

Example. In the case of 2442, I ran the multi-agreeing algorithm two ways. The first used the lexicographic ordering of the links. This ordering required 9 minutes and 45 seconds to find the solution. The second ordering used my methodology discussed in Chapter 6. This took 2 seconds to find the solution. This is an improvement of three orders of magnitude. A similar improvement occurred in other cases.

Hypothesis: At least one agreeing order is more efficient than others. Efficiency is defined as less time consuming. There exists a most efficient way to agree the links.

An exhaustive check of all possible orders is not feasible, given that the number of links ranges from 10 to 200 for the various small scale variants of AES tested in this research. Therefore, various strategies are created and experimented with in this research. The goal is to find efficient orderings which minimize the number of new equations (degrees of freedom) added with each increase of the agreed link set. Three strategies are discussed in this section.

Strategy #1. Begin by picking a start link and a destination link. Then use the other links to introduce the missing equations from the destination link into the agreed set. Once all these

equations are in the agreed set, this enables the destination link to be a check-for-consistency link. The refinement of the idea was to pick two links (as the start and destination) that were similar in equation content in the hope that the check-for-consistency could occur after only a few additional links were added.

In AES, the equations featuring the later state arrays were not as prolific in the link structure. This observation influenced the strategy. In fact, for many of the later equations, there were only two links (from the default link set) that use them. These became the start and destination links. The start link was designated as the link that used the least number of equations in it. The ordering of the middle links was focused on bringing in the missing equations so that a check-for-consistency was quickly reached with a minimal increase to RHS set size. The issue with this strategy is that it was difficult to transition from the start to destination link without adding multiple new equations with each link. To counter-balance this, sometimes dependent links could be created to assist. While dependent links helped reduce the size of consistent RHS sets to check, it increased the number of links to process within the program. Additionally, it was difficult to find dependent links that were actually helpful.

link number	equations in link	new eqns to order	# degrees of freedom
Link #8	1,5,9	1,5,9	2
Link #0	0,5	0	2
Link #5	1,3,7	3,7	3
Link #9	0,1,3,9	none	2
Link #7	0,1,3,8	8	2
Link #6	0,1,4,8	4	2
Link #4	0,2,3,7	2	2
Link #1	0,1,2,3,4	none	1
Link #2	0,2,6	6	1
Link #3	0,1,2,6	none	0

Table 5.2: Example of strategy #1.

Table 5.2 gives an example of an ordering using strategy #1 with the default set of links for 214e. The links are listed in the order of multi-agreeing. The first column gives the link reference number from the process of creating the default set of links. The second column lists the MRHS equation numbers that are part of the specific link. The third column denotes which of the equations are new to the ordering with the addition of the link. The last column gives the count of the degrees of freedom once the link is added to the agreeing ordering. For this example the inclusion of Equation #9 defined the start and destination links, Links #8 and #9, respectively.

This strategy worked with small *nrce* varients of AES. It failed horribly on large variants of AES due to the long processing times. This strategy also showed how important the key variables and key schedule equations were to the various links. Even with its shortcomings, this strategy did find the solution to the system faster than by just using the numerical order of default links.

Strategy #2. Begin with the link that has the bulk of the key equations in it. Those equations (and their respective variables) affect the other equations the most and produce the most relationships amongst the other equations. Once the start link is chosen, then use the other relationships between variables to see where the quickest check-for-consistency could occur amongst the state equations. Based on the structure of AES, this would occur in the equations representing the later state arrays. Table 5.3 gives an example of an ordering using strategy #2 with the default set of links for 214*e*. This is the same case as shown in Table 5.2.

link number	equations in link	new eqns to order	# degrees of freedom
Link #1	0,1,2,3,4	0,1,2,3,4	4
Link #0	0,5	5	4
Link #2	0,2,6	6	4
Link #3	0,1,2,6	none	3
Link #5	1,3,7	7	3
Link #4	0,2,3,7	none	2
Link #6	0,1,4,8	8	2
Link #7	0,1,3,8	none	1
Link #8	1,5,9	9	1
Link #9	0,1,3,9	none	0

Table 5.3: Example of strategy #2.

This strategy produced better results than the first strategy. However, there were still flaws. Using a large first link (meaning it causes a large number of RHS sets) did not take into consideration the structure of the equations. It focused on the quantity of equations instead of the quality of relationships between the equations.

Strategy #3. In an effort to better explore the relationships between the links, Strategy #3 was created: an agreeing tree. The purpose of the tree was to choose the cheapest (in terms of the introduction of free equations) sequence of links to agree.

This strategy consists of three parts: 1) determine the root of the tree, 2) create the agreeing tree, and 3) search the tree. Structurally, this tree consists of nodes and weighted, directed edges. The nodes represent links. The edges are weighted based on the number of free equations that are

added to the order when agreeing one link to the next. Also, these are directed edges because order matters.

Part I of this strategy is to determine the root of the tree. This is based on how cheaply the starting link (tree root) can move to the next link. Recall that we are ultimately looking for an ordering that minimizes the number of free equations added with each link. During testing of larger *nrce* variants, it was found that a threshold on the size of the starting link was also necessary. For illustration purposes, the 222*e* case is shown in this section.

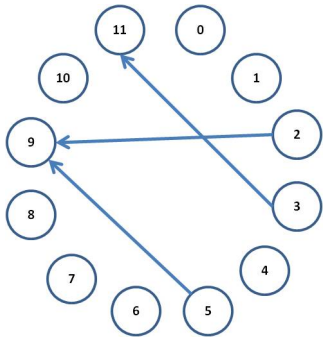

Figure 5.5: Determine root of the agreeing tree.

Figure 5.5 shows the starting possibilities. Each node is a multi-link. The edges in the graph represent an order that results in one free equation being added. Therefore, it suggests that either Link #2, #3 or #5 should be the root. For purposes of this explanation, Link #3 is chosen.

Part II of this strategy is to create the agreeing tree. The tree starts with the root node. Different branches are added that correspond to options in an agreeing order. Due to the fact that minimization of the number of new equations at each link is the goal—thresholds are placed on the branches. Namely, the first option is to only use paths that have one free equation. This worked to create the agreeing tree for 222*e*. However, when creating the agreeing tree for 244*e*, branches terminated early. Therefore, links that added two free equations were allowed to be considered in larger cases.

Tree construction continues until the leaf node is reached (i.e., the path has every possible link in it). The overall value of this path (sum of the weights of its edges) is its utility score. The lower the score, the better. Once the first path is successfully created and its utility score calculated, this becomes another threshold on the tree. This allows other paths to be terminated early if they were going to reach a utility score larger than the current best. While I perform these

calculations manually, they could be easily run on parallel processors for larger variants.

The 222*e* case is continued for illustration purposes. Figure 5.6 shows the selection of the root node. The nodes on the left of the arrow show the options for tree branches. The graph on the right of the arrow shows the resulting agreeing tree.

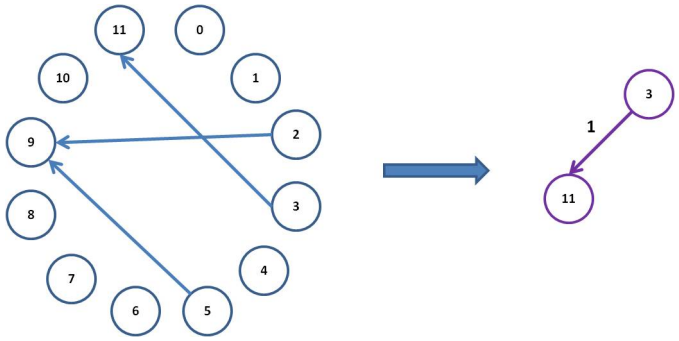

Figure 5.6: Step 1 of agreeing tree.

Now that Link #3 and #11 are used in the agreeing tree, this creates a new "super" node when examining our next possibilities. This is the node N on the left hand side of Figure 5.7. This graph on the left now shows all the current link orders that would result in one free equation. There are four options at this point (shown as four directed edges). As this example is for illustration purposes, we focus on the paths that link the node N to other links instead of starting a new tree. Therefore, there are only two options (branches) for the tree (Link #0 or Link #1). This is shown on the right hand side of Figure 5.7.

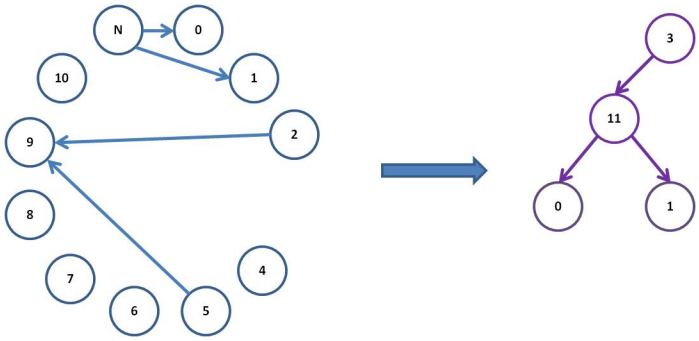

Figure 5.7: Step 2 of agreeing tree.

The 222*e* example continues down the left-most branch of the tree. Now, there is only one option for a link that results in only one free equation being added. Therefore, Figure 5.8 follows easily.

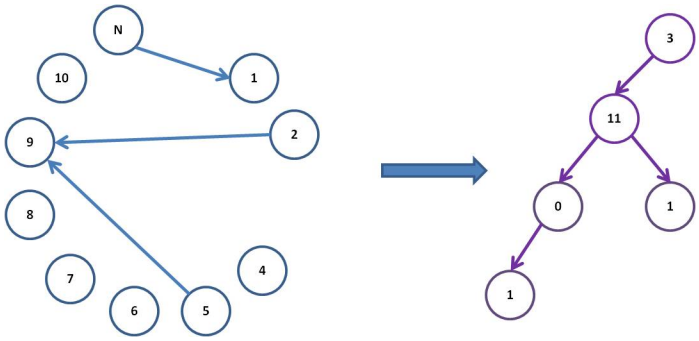

Figure 5.8: Step 3 of agreeing tree.

At this point, there are two options for the continued construction of the agreeing tree. Figure 5.9 shows the options as Link #2 and Link #10.

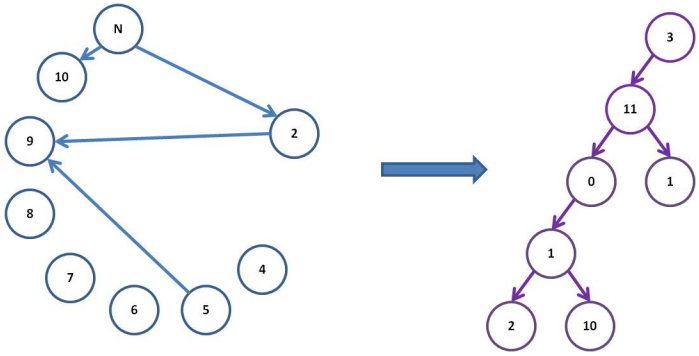

Figure 5.9: Step 4 of agreeing tree.

This construction process continues until all possible branches are created from using Link #3 as the root node of the agreeing order.

Part III of this strategy is to search the tree. This part is fairly trivial once the tree is constructed. In fact, depending on how the algorithm is programmed, this could be done in parallel with the tree construction depending on available resources. Regardless of when this part of the strategy is actually executed, common search algorithms can be used. They traverse the various paths to determine 1) if it was a full path (had all the link nodes—therefore, would be a valid ordering) and 2) its utility score. To ensure that edges denoting one free equation were significantly better than two, a weighting factor is used. An example of a weight scheme is to allow paths with one free equation to have a value of one, paths with two free equations to have a value of ten, paths with three free equations to have a value of one hundred, and so on. Once the smallest utility

value is found, the agreeing order is set.

5.5.2 Structure

Ultimately, the structure of AES should guide this research to an efficient order of multi-links to use in the multi-agreeing process. This is because the MRHS equations are not a randomly generated set of equations, but are based on the AES algorithm which is known. Therefore, structure should help.

In this research, I analyze small variants of AES to look at the structure of the symbols, their resulting links, and how these things might affect the agreeing order.

What does the number of rounds do to the AES structure?

The two-round version of AES ($n = 2$) fails to hold some of the algebraic properties of ten-round AES. In fact, it always has interesting side effects and relationships that make solving the system easier than other round totals. This is because in the two round version of AES: 1) the Mix Column function is only used once, 2) equations only involve variables from one state array instead of multiple ones, and 3) some of the link equations provide a relationship between the first state array and the last one. Namely, it links the plaintext to the ciphertext directly. Because of these unique relationships, two-round versions of AES are interesting but not extremely helpful in determining an ordering for multi-agreeing links of larger round AES variants.

The three-round version of AES begins to showcase all the features of 10-round AES. In general, once an efficient agreement order is determined for three rounds, the concept scales up to ten rounds. Efficiency is defined by minimizing the number of free variables in the ordering. The only exception to this observation was the case of 3244. In fact, I found a speedy ordering to 3244 that could not scale up to 4244 and beyond.

Based on this anomaly, I choose four rounds of AES as the minimum n to create an agreeing order for before I attempted to scale to ten rounds. Four rounds was still very manageable to gather good insights on how the various symbols and variables interact with each other while efficiently looking for patterns. The ordering found for four rounds is carefully scaled to ten rounds to ensure time comparisons could be made. Table 5.4 provides a snapshot of the size of systems this research explores.

n	# MRHS eqns	# variables
2	40	40
3	60	60
4	80	80
10	200	200

Table 5.4: $n44e$ AES system size.

What does word size do to the structure?

The options for field size in AES are 2^2, 2^4, and 2^8. This corresponds to words of length 2 bits, 4 bits or 1 byte. I use $e = 2$ to perform calculations by hand while developing the methods and analyzing patterns. The only issue with the 4-element field is that all possible S-boxes are affine. While not changing the matrices A or the actual multi-links, an agreeing order in this field will delete RHSs sooner and achieve a solution faster. Therefore, it can skew the expected results for larger fields. In this research, once an ordering for a *nrc* variant is found, it was run on $e = 2$ to to test the ordering for correctness and speed prior to executing on the larger fields.

The only difference between $e = 4$ and $e = 8$ is that in $e = 4$, blocks of the state array and key matrix are presented by one hexadecimal character while in $e = 8$, blocks require two hexadecimal characters to hold their value. Everything else is identical with respect to the relationships and structure of the system. Therefore, the most efficient ordering for multi-agreeing is the same. This is quite helpful in analyzing AES as testing could be realistically performed on orderings of links in the 16-element field using current computers in a realistic amount of time which would take too long in the field of 256 elements.

How does a "tall and skinny" state array affect the structure?

A tall and skinny state array occurs when $r > c$ in a variant of AES. In this research, $n41e$ and $n42e$ small scale variants of AES are analyzed. This shape also showcases the affect of the Mix Columns function on the structure. Ultimately, an increased percentage of the MRHS equations for these systems are from the key schedule algorithm. Therefore, more key variables are present in the links in general. The links are also more sparse of equations. The links from $n41e$ have between two and six equations in each link with an average of 3.5 equations per link. The links from $n42e$ have between two and fourteen equations in each link with an average of 6.5 equations per link. The sparseness of the links and greater number of MRHS equations to describe these systems, make them slower to solve than their "short and fat" counterparts.

How does a "short and fat" state array affect the structure?

A "short and fat" state array occurs when $c > r$ in a variant of AES. In this research, $n14e$ and $n24e$ small scale variants of AES are analyzed. This shape also showcases the affect of the Shift Rows function on the structure. Ultimately, an increased percentage of the MRHS equations for these systems are from the encryption algorithm. Therefore, more state array variables are present in the links in general. There are also fewer total equations (and therefore links) for this shape. The links from $n14e$ have between two and eleven equations in each link with an average of 4.6 equations per link. The links from $n24e$ have between two and thirteen equations in each link with an average of 5.5 equations per link. This shape routinely solves itself faster than the "tall and skinny" AES shape.

How does a square state array affect the structure?

A square state array occurs when $r = c$ in a variant of AES. In this research, $n22e$ and $n44e$ small scale variants of AES are analyzed. The links from $n22e$ have between two and thirteen equations in each link with an average of 5.4 equations per link. The links from $n44e$ have between two and fifteen equations in each link.

Finally, it is interesting to note that, contrary to the results obtained by Raddum and Semaev, there does not always exist an unique solution to the AES MRHS system of equations. Their technique (revolving around guessing to determine the final answer) yields an answer—but gives no indication if another answer also exists. Our method of extending agreeing and linking into multiples, completely solves the system, and provides *all* of the solutions upon completion.

The insights gained from all these strategies and AES's structure led to the creation of an agreeing order scheme for the various *nrce* variants of AES. The orderings are discussed in Chapter 6 along with the time and solution results.

CHAPTER 6:
Results

Small scale variants of AES showcase the algebraic properties of full AES. By understanding the relationships between the transformations, you can create an efficient ordering of multi-links for processing in the multi-agree algorithm. This chapter gives the timing results and insights determined from exploring *nrce* variants of AES. All AES results are computed on a 2.2 GHz AMD processor with 4 GB of memory. This chapter also presents other researchers' timing results for a comparison on small scale variants. This chapter then explains the model we created to estimate the time needed to find a solution in larger variants of AES (to include A448). Finally, this chapter presents the results from attempting to use multiple plaintext/ciphertext pairs to more quickly find solutions.

6.1 Results of New Method

Ultimately, it came down to looking for and identifying patterns within the data. Two different mappings were useful in determining an agreeing order. The first is a mapping of multi-links to equations for a given *nrce*. This assists in visualizing free equations and link relationships. Table 6.1 gives an example of this mapping for the 2144 case. The 'x' denotes that the equation is present in the multi-link.

	equations									
links	0	1	2	3	4	5	6	7	8	9
0	x					x				
1	x	x	x	x	x					
2	x		x				x			
3	x	x	x				x			
4	x		x	x				x		
5		x		x				x		
6	x	x			x				x	
7	x	x		x					x	
8		x				x				x
9	x	x		x						x

Table 6.1: 2144 links - equations mapping.

The second useful mapping is a mapping of variables to links. Each multi-link uses an equations' top row (input to S-box), bottom row (output of S-box), or both rows to express the de-

pendence relationship. Various variables are involved in each row in the equation. This mapping denotes which variables are actively used in the link and which variables came along because they are in the unused row of the link's equations. This is important because these variables (used or not used in a link) can influence the remainder of the multi-agreeing order. Table 6.2 gives an example of this mapping for the 2144 case. The '•' denotes that the variable is used in the link and the '○' denotes the variable is in an unused row of the link's equations. These two visualizations, along with an understanding of the structure of the system of equations, enables an ordering to be selected.

	variables									
links	0	1	2	3	4	5	6	7	8	9
0	○			•	○					○
1	•	•	•	•	•	○	○	○	○	
2	•			○	•	○	•			
3	•	○	○	○	•	•	○			
4	•	•		○	•	○	○	•		
5		•	○	○	•	•		○		
6	○	•	•	•	•	○			•	
7	○	•	•	•	•	•		○	○	
8		•	•	•	•	○				•
9	○	•	○	•	•	•		○		○

Table 6.2: 2144 links - variables mapping.

The results in this section are organized by the shape of the AES state array (defined by *rc* variables). The different shapes require different ordering schemes to solve the system. In general, once an ordering is found for four rounds of the shape, it is easily scaled up to the ten round version of the shape. In the following cases, I describe the final method used to find the key. Many different orderings were attempted to reach the final results. See Appendix D for the C code that implements our methods on AES cases.

*n*14*e*

Using the visualization from the two mappings and an understanding of the AES structure, a link ordering is created for *n*14*e*. There are four interesting observations about the shape of this state array. The first observation is that a two round version of this shape is very different from the other round versions. This confirms the uselessness of watching $n = 2$ operate. Therefore, the focus is on creating an efficient ordering for $n > 2$ rounds instead. The second interesting observation is that only one link used variable #0. This is a key variable so it is important to get into the ordering. The link with this variables is first used as a possible start point for the

multi-agreeing order. However, there are too many free equations within this link to create an efficient multi-agreeing order. The consequence of this observation is that any viable ordering must have the other equations (in this link) in the agreed set prior to including this link. This enables the number of consistent RHS sets to not increase. The third observation is that all the links with state array equations also had multiple key equations in them. This reinforces the concept that we have to introduce the key equations into the multi-agree ordering first before bringing in the least-used state array equations. The final interesting observation is that there are no groupings of sequential state array equations in the multi-links because the Shift Rows transformation has no impact on this shape. These groupings of sequential equations increase the complexity of finding an efficient multi-agreeing order (as will be seen in the other AES shapes).

Recall that the multi-agreeing order focuses on minimizing the number of free equations introduced into the order with the inclusion of each multi-link. In determining a multi-agreeing order for this shape, I use a ceiling of four degrees of freedom. This is equal to the number of initial key variables in this shape ($r \cdot c$). This corresponds to 65,536 consistent RHS sets with a 4-bit word. This number is chosen based on how large the number of RHS sets grow with an 8-bit word.

Various orderings of multi-links were tested on the smaller n variants of this shape. Once a likely ordering candidate is found, it is scaled up to larger n cases. This is done by matching multi-links from one case to the next using the mapping of links to variables. Once the matching is complete, the new n case has its links placed in the same relative order as the smaller n case. This is extremely helpful as it enables the bulk of testing to be conducted on cases that are small enough to quickly run on the computer. Throughout this research, if an ordering is created for an AES variant with $n \geq 4$, then it can be successfully scaled up to $n = 10$.

The final ordering methodology for this AES variant utilizes a starting link with only key variables in it. Additionally, this starting link has to contain fewer equations than the link with variable #0 in it. There are only three links that match this criteria within the $n14e$ shape. All possible orderings of these links are tested to find the best results. The final efficient order brought these three links in first, which resulted in Equations #0 through #7 being in the agreed set. By ***agreed set*** it is meant that the equations exist in a multi-link in the current order. This means that their information and structure can be used to bring the next multi-link into the order. The next multi-links are placed in the multi-agreeing order in such a fashion that the remaining

free equations are introduced in numerical order (8, 9, 10, ...). The agreeing order of multi-links for the case of 3144 is shown in Table 6.3 to assist the reader in their understanding of this process.

link number	new eqns to agreed set	# degrees of freedom	comment
Link #4	0,1,2,3,5	4	link with only key variables in it
Link #0	4	4	
Link #2	6	4	
Link #6	7	4	
Link #1	8	4	
Link #3	9	4	
Link #5	10	4	
Link #7	11	4	
Link #8	–	3	check-for-consistency
Link #9	12	3	
Link #10	–	2	check-for-consistency
Lnk #11	13	2	
Link #12	–	1	check-for-consistency
Link #13	14	1	
Link #11	–	0	solved with this check

Table 6.3: Agreeing order for 3144.

Utilizing this ordering concept, the multi-agree function is triggered four times in the program. Recall that this function is triggered when no new equations are added with a multi-link, resulting in a check-for-consistency. Of the four, only the last two multi-agrees delete RHSs. Therefore, I suppress the program from executing the first two. This saves time and did not affect the completion times of the multi-agrees that did delete RHSs. That left two multi-agrees to execute. The first one deletes approximately 34% of the RHS and then the second one removes the remainder of the non-solutions. Table 6.4 displays the *nrce* value, the number of solutions found, and the amount of time to solve these variants. While the program correctly finds the solution each time, in some cases additional solutions are also found which indicates that, under

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
2142	1	0.001s	2144	3	0.003s	2148	4	10.461s
3142	1	0.001s	3144	3	0.019s	3148	2	20m 54s
4142	1	0.001s	4144	1	0.051s	4148	1	53m 58s
A142	1	0.004s	A144	2	0.209s	A148	4	3h 47m 3s

Table 6.4: $n14e$ AES results.

certain conditions, multiple keys can take a single plaintext and turn it into a single ciphertext. This is a very interesting result.

*n*24*e*

Now that an efficient agreeing order for *n*14*e* is found, this assists in our creation of an agreeing order for *n*24*e*. The major difference in this AES shape is that there are now pairs of sequential equations in the multi-links that come from sequential state array variables. Table 6.5 illustrates the two pairs of sequential equations (Equations #8 and #9, Equations #10 and #11). This is because the Shift Rows transformation now manipulates this shape. Luckily, for the construction of our agreeing order, this shape also has multi-links that bring in each pair of the sequential equations separately. In Table 6.5, it clearly shows that Link #3 has Equation #8 without Equation #9 and Link #1 has Equation #9 without Equation #8. However, these links also include many of the lower-numbered equations (which have many key variables). Therefore, the multi-agreeing order concept for *n*24*e* is to first bring in the low-numbered equations (focused on the most prolific ones). For example, in Table 6.5, these would be Equations #0 through #7. The most prolific equations in this shape are Equation #0 and #2 which are each in the majority of the links. Once the first eight equations are introduced into the agreed set via the multi-agreeing order, the state array equations are brought into the agreed set separately. In Table 6.5, these would be Link #3 to bring in Equation #8, Link #1 to bring in Equation #9, Link #10 to bring in Equation #10, and then Link #2 to bring in Equation #11. This process minimizes the number of free equations associated with each multi-link.

However, the problem remains to establish the beginning of the agreeing order which brings Equations #0 through #7 into the agreed set. I tried quite a few different combinations to pick the best starting multi-link for the process. Experimentation shows that the best multi-link to start the multi-agreeing order with was a link of five equations. Then, all the links with one free equation are added to the order until only links with two or more free equations are left. Therefore, the number of consistent RHS sets has to increase. However, the next most efficient multi-link to agree has three free equations (causing a large increase in the number of consistent RHS sets). This link is specifically chosen because it incorporated the two most commonly used equations. Ultimately, it proves more efficient to add these three free equations with this multi-link, than to use a less dense multi-link next in the agreeing order. Eventually, the links with one free equation are exhausted and a link with two free equations joins the agreeing order. This multi-link is chosen because it uses the last key equation missing from the agreed set. The remainder of the multi-links fell into the agreeing order without increasing the consistent RHS

	equations																			
links	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0		x									x									
1	x									x										
2	x	x	x			x		x				x								
3	x	x		x	x		x		x											
4	x				x	x							x							
5	x		x		x								x							
6	x		x				x	x						x						
7	x		x	x		x		x						x						
8	x				x		x	x							x					
9			x				x								x					
10	x		x						x	x						x				
11				x		x										x				
12		x		x					x	x							x			
13		x	x	x			x										x			
14			x								x	x						x		
15	x		x	x		x												x		
16				x							x	x							x	
17		x	x				x												x	
18	x		x		x	x		x												x
19	x			x		x														x

Table 6.5: 2244 links - equations mapping.

set count. This concept results in eight degrees of freedom (note that $r \cdot c = 8$ for this shape).

To scale this ordering from n to $n+1$, the mappings of links to variables is examined. Each link has an unique profile in this table. Therefore, it is fairly straightforward to scale this ordering from n-to-n using this information. This also ensures that the same concept is used for a specific rc shape so that timing comparisons can be made in reference to the number of rounds and the field size.

With this ordering concept, the multi-agree function is triggered six times in the program. Of the six, only the last two multi-agrees delete RHSs. Therefore, I suppresses the program from executing the first four. That leaves two to execute. The first one deletes an average of 28% of the RHS from the equations. This deletion percentage ranges of 21-33% across this shape variant. The second multi-agree removes the remainder of the non-solutions. Table 6.6 displays the results. There are no listed times for the $e = 8$ versions of this shape. This is because they take longer than 300 hours to run on a single processor. Estimations for their run time is

discussed in Section 6.2 and listed in Table 6.26.

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
3242	1	0.019s	3244	2	13m 20s	3248	$\star$	$\star$
4242	4	0.090s	4244	2	1h 27m 33s	4248	$\star$	$\star$
5242	2	0.110s	5244	6	1h 48m 31s	5248	$\star$	$\star$
6242	2	0.166s	6244	2	2h 49m 36s	6248	$\star$	$\star$
7242	4	0.220s	7244	3	3h 39m 40s	7248	$\star$	$\star$
A242	4	0.258s	A244	2	7h 11m 40s	A248	$\star$	$\star$

$\star$ No solution found within 300 hours of computation time.

Table 6.6: $n24e$ AES results.

$\underline{n41e}$

Using the visualization from the two mappings and an understanding of the AES structure, a link order is created for $n41e$. Three interesting observations are deduced from this AES shape. The first observation is that the size of the groupings of state array equations is equal to the number of rows in the state array. Since $r = 4$ in this shape, there are groupings of four sequential state equations in the links. In Table 6.7, the reader can see the first quad of sequential equations consists of Equations #12, #13, #14, and #15. The second quad consists of Equations #16, #17, #18, and #19. Like the previous case ($n24e$), there are also links that include each of these quad equations independently. In Table 6.7, Link #0 has Equation #12, Link #2 has Equation #13, Link #3 has Equation #14 and Link #4 has Equation #15. These are the equations in the first quad.

The second observation is that the links for $n41e$ are more equation sparse than the links for previous shapes. Therefore, there is less information per link in this *nrce*. The consequence is that it takes more links to reach a check-for-consistency. This is why this shape is slower to solve than the equivalent $n14e$ cases.

The third observation is that the links are more intertwined than previous shapes. The mapping of links to equations for the 3414 AES variant is shown in Table 6.7. Notice that the links with the single equations from the second quad have the entire first quad in them. For example, Link #8 has Equation #16 without its quad. However, Link #8 also has Equations #12-15 (first quad) in it. Therefore, a viable agreeing order introduces the first quad of equations as soon as possible, definitely prior to the other state equations. In this example, Equations #12-15 need to enter the agreed set as quickly as possible. Therefore, Links #0, #2, #4, and #6 are toward the beginning of the agreeing order.

	equations																							
links	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0				x									x											
1						x			x															
2	x													x										
3				x			x																	
4		x													x									
5							x			x														
6			x													x								
7			x			x																		
8	x												x	x	x	x	x							
9								x			x													
10		x											x	x	x	x		x						
11	x							x																
12			x										x	x	x	x			x					
13					x							x												
14				x									x	x	x	x				x				
15		x			x																			
16					x												x	x	x	x	x			
17									x												x			
18						x											x	x	x	x		x		
19										x												x		
20							x										x	x	x	x			x	
21											x												x	
22								x									x	x	x	x				x
23												x												x

Table 6.7: 3414 links - equations mapping.

For all sizes of n, the first set of links in the multi-agreeing order are Link #0, #2, #4 and #6. Each of these links have two equations in them—one of the state equations and one of the key equations. After starting the order with these four links, the first quad of equations is in the agreed set along with the first four key equations. The remaining equations are introduced into the agreed set in numerical order. Namely, based on Table 6.7, the agreeing order continues with Link #15 (Equation #4), Link #7 (Equation #5), Link #3 (Equation #6), Link #11 (Equation #7), etc. Each of these links brings only one new equation into the order and therefore, does not increase the RHS set count. The ordering results in a maximum of four degrees of freedom.

With this ordering concept, the multi-agree function is triggered four times in the program. Of the four, only the last two multi-agrees delete RHSs. Therefore, I suppress the program from executing the first two. That leaves two to execute. The first one deletes approximately 36% of the RHS and then the second one removes the remainder of the non-solutions. Table 6.8 shows the results.

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
3412	4	0.002s	3414	1	0.052s	3418	5	50m 22s
4412	1	0.002s	4414	1	0.079s	4418	2	1h 21m 10s
5412	4	0.003s	5414	2	0.062s	5418	2	1h 11m 43s
A412	1	0.006s	A414	2	0.267s	A418	1	4h 29m 51s

Table 6.8: $n41e$ AES results.

$\underline{n42e}$

Now that an efficient agreeing order for $n41e$ is found and lessons learned, this assists in our creation of an multi-agreeing order for $n41e$. In this shape, there are groups of four sequential state array equations in the multi-links due to $r = 4$. There are also links with each quad equation separate but these links are scattered throughout the default link structure. See Table 6.9 for a portion of the mapping of links to equations in the 3424 case. Notice that the first quad uses Equations #12-15. Now, Link #15 has Equation #12 as a single, Link #1 has Equation #13, Link #5 has Equation #14 and Link #9 has Equation #15. The other major difference in $n42e$ compared to $n41e$ is that some of these singles are tied to more than one key equation. Therefore, quick and easy orderings are not possible. For example, there are three key equations in Link #5 and three in Link #15. In general, the multi-links that contain the singles of the quads also include one to three key equations with them.

	equations																										
links	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	25	26
0		x																	x								
1	x													x														
2			x	x			x													x								
3		x					x			x																		
4				x				x					x	x	x	x					x							
5		x	x			x									x													
6					x												x	x	x	x		x						
7				x	x							x																
8		x				x							x	x	x	x							x					
9			x													x												
10							x										x	x	x	x				x				
11	x					x			x																			
12								x									x	x	x	x					x			
13				x													x											
14	x				x								x	x	x	x										x		
15	x			x				x					x															
16						x											x	x	x	x							x	
17	x	x			x													x										
18			x				x						x	x	x	x												
19			x					x			x																	
20				x	x			x													x	x	x	x				
21				x	x			x	x																			
22	x					x																			x	x	x	
23	x				x	x				x																		
24		x	x	x																	x	x	x	x				
25		x					x				x																	
26			x					x																	x	x	x	

Table 6.9: Portion of the 3424 links - equations mapping.

In addition, very few links have a multitude of the key variables in them. While the $n41e$ multi-links reference multiple key variables within them, the $n42e$ multi-links only use one to two key variables directly and reference one to three more. Finally, the relative density of equations in the links is significantly less compared to $n41e$. Therefore, it takes longer (with numerous free equations) to bring all the key variables into the agreeing order. The execution times in Table 6.8 and Table 6.10 showcase this difference.

The $n42e$ ordering of multi-links begins by focusing on the first quad of equations. This is because like the $n41e$ case, the singles for future quads depend on the first ones. In order to minimize degrees of freedom, the order begins with a link that has a single from the first quad. In Table 6.9, options are Link #1, #5, #9 or #15. This research analyzed different combinations of starting links for the different singles. The chosen starting link is the one that minimizes the number of free equations while maximizing the number of key equations. This ordering continues until all the links with one free equation are used. Then, a link with four free equations is brought into the agreeing order. This link is chosen in order to get all of the first quad equations into the agreed set. Other options for the next link (with fewer free equations) are also attempted. However, in the end, they do not save time, as the number of free equations eventually grows to match the four brought in.

Then, links with one free equation are added to the order. However, this is still not enough to bring the second quad in the agreed set or to reach a check-for-consistency at the current number of degrees of freedom. Therefore, a link with two free equations is added in order to bring in the second quad. Then, a link with two free equations is added to bring in the last key equations. Once all the key equations and first two quads are in the agreeing order, the remainder of the links are added to the order in a way that did not increase the number of consistent RHS sets. Once again, the number of key variables ($r \cdot c$) is equal to the number of degrees of freedom in the multi-agreeing order.

With this ordering concept, the multi-agree function is triggered eight times in the program. Of the eight, only the last two multi-agrees end up deleting RHSs. Therefore, I suppress the program from executing the first six. That leaves two to execute. The first one deletes an average of 36% of the RHS from the equations. This deletion percentage ranges from 24-50% across these variants. The second multi-agree removes the remainder of the non-solutions. Table 6.10 shows the results. Just like the $n24e$ AES variants, the cases when $e = 8$ are not able to complete in a reasonable amount of time. See Section 6.2 for completion time estimates.

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
3422	4	0.032s	3424	1	49m 54s	3428	⋆	⋆
4422	2	0.091s	4424	1	1h 26m 17s	4428	⋆	⋆
5422	16	0.397s	5424	3	2h 32m 6s	5428	⋆	⋆
6422	2	0.219s	6424	3	3h 45m 53s	6428	⋆	⋆
7422	1	0.310s	7424	1	5h 11m 48s	7428	⋆	⋆
A422	4	0.583s	A424	3	10h 7m 45s	A428	⋆	⋆

⋆ No solution found within 300 hours of computation time.

Table 6.10: *n*42*e* AES results.

*n*22*e*

Using the visualization from the two mappings and an understanding of the AES structure, a multi-agreeing ordering is determined for *n*22*e*. Unlike the previously discussed AES rectangles, this is the first square shape. The idea is to use the insights and orderings from the rectangular shaped AES versions to help create an efficient ordering for *n*22*e*. See Table 6.11 for the mapping of links to equations in the 3224 case.

	equations																	
links	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	x	x	x							x								
1		x	x			x												
2		x		x			x	x			x							
3	x							x										
4			x						x	x		x						
5	x			x	x													
6				x					x	x			x					
7		x							x									
8	x		x				x	x						x				
9	x	x		x			x											
10		x	x	x							x	x			x			
11		x	x	x	x										x			
12	x			x									x	x		x		
13	x		x	x		x										x		
14		x	x										x	x			x	
15				x	x												x	
16	x		x	x							x	x						x
17			x			x												x

Table 6.11: 3224 links - equations mapping.

As is predicted from the other shapes, the state equations are in pairs in the various multi-links. For instance, the first pair of sequential state array equations in 3224 are Equations #6 and #7. Additionally, it is observed that the multi-links with the earlier sequential pairs have both key equations and later state equations in them. As seen in Table 6.11, Link #2 and #8 include the first pair. Both of these multi-links also include key equations and state equations from later rounds of the AES algorithm. Therefore, introducing key equations and the first pair of state equations efficiently within the multi-agreeing order is the goal. Like other *nrce* cases, the $n = 2$ case is not helpful in deducing an ordering for increasing n sizes.

Based on this research, the best starting link for this shape is one with three key equations and one state equation from the second pair. The next multi-link in the agreeing order is one with an additional key equation and one state equation from the first pair. This gave a total of four degrees of freedom for the ordering. These two multi-links introduce the four most prolific key equations into the agreed set which make subsequent decisions easier. Prolific for the $n22e$ shape means that these four key equations are more than twice as common as the other key equations in the links. Based on relationships, the remainder of the linking order does not increase the number of consistent RHS sets. Basically, the remainder of the multi-links are added to the agreeing order in such a way that the state equation pairs are brought in as fast as possible in order to get to the later state equations and multi-links where the checks for consistency occurr. Table 6.12 gives the agreeing order for 3224 based on the above description. The first column gives the link number as defined from the default set of links. The second column lists the new equations that are brought into the agreed set from the link. The third column displays the number of degrees of freedom once the multi-link is added to the multi-agreeing order. The final column explains when the pairs of sequential state equations are in the agreed set and when a check-for-consistency link occurs.

With this ordering concept, the multi-agree function is triggered four times in the program. Of the four, only the last two multi-agrees delete RHSs. Therefore, I suppress the program from executing the first two. That leaves two to execute. The first one deletes approximately 37% of the RHS and then the second one removes the remainder of the non-solutions. Table 6.13 shows the results.

link number	new eqns to agreed set	# degrees of freedom	comment
Link #0	0,1,2,9	3	
Link #9	3,6	4	
Link #3	7	4	1st equation pair in
Link #7	8	4	2nd equation pair in
Link #2	10	4	
Link #4	11	4	3rd equation pair in
Link #5	4	4	
Link #10	14	4	
Link #11	–	3	check-for-consistency
Link #1	5	3	
Link #16	17	3	
Link #17	–	2	check-for-consistency
Link #6	12	2	
Link #8	13	2	4th equation pair in
Link #12	15	2	
Link #13	–	1	check-for-consistency
Link #14	16	1	
Link #15	–	0	solved with this check

Table 6.12: Agreeing order for 3224.

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
3222	2	0.000s	3224	2	0.030s	3228	1	28m 48s
4222	2	0.001s	4224	2	0.054s	4228	1	55m 56s
5222	4	0.003s	5224	1	0.085s	5228	1	1h 28m 59s
6222	2	0.001s	6224	3	0.087s	6228	2	2h 4m 50s
A222	2	0.004s	A224	2	0.291s	A228	2	5h 17m 38s

Table 6.13: *n*22*e* AES results.

*n*44*e*

Using all of the experimental data from this research and the heuristics developed, a multi-link ordering is determined for *n*44*e*. The mapping of the links to equations for 3444 starts on page 74 due to its size. As is predicted from the previous results, the state equations exist in sequential quad groups within the multi-links. For example, Equations #12, #13, #14, and #15 are the first quad in the 3224; Equations #16, #17, #18, and #19 are the second quad; and so forth. The multi-links with these quad equations also include one to two later quad singles and a multitude of key equations within them. For example, see Link #10, #16, #21, #22 which include the first quad of equations. These relationships cascade through the multi-links. Therefore, once again, the earliest quads of equations must be included in the agreeing order prior to the later ones in

order to minimize the degrees of freedom. This is clearly shown in the mapping of 3444 as the four singles from the 5th quad require the 1st, 2nd, 3rd, and 4th quad to already be in the order. This is the same requirement for the 6th, 7th and 8th quads—that the first four quads have to already be in the order to minimize the order's degrees of freedom.

Therefore, the singles from the early quads need to join the agreeing order early. This implies that certain key equations need to be present in the order first. This research focuses on creating an efficient order that would bring these key equations in first while minimizing the degrees of freedom. Based on these requirements, an agreeing order is found that (at its peak) has sixteen degrees of freedom. It is not possible to reduce this number based on the sparse nature of the MRHS equations for AES using the default set of links. Sixteen is the $r \cdot c$ value for this AES shape.

With this ordering concept, the multi-agree function is triggered sixteen times in the program. Of the sixteen, only the last two multi-agrees delete RHSs. Therefore, I suppress the program from executing the first fourteen. That leaves two to execute. Table 6.14 shows the results for the 4-element field. The larger fields take computing time than is feasible on a single processor. The only exception to this is the 2444 case which is always an abnormality. Additionally, the agreeing order for $A44e$ is located in Appendix B.

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
2442	1	0.228s	2444	4	21h 37m 2s	2448	⋆	⋆
3442	2	54m 4s	3444	⋆	⋆	3448	⋆	⋆
4442	1	2h 53m 20s	4444	⋆	⋆	4448	⋆	⋆
5442	1	3h 11m 47s	5444	⋆	⋆	5448	⋆	⋆
A442	2	17h 28m 48s	A444	⋆	⋆	A448	⋆	⋆

⋆ No solution found within 300 hours of computation time.

Table 6.14: $n44e$ AES times.

	equations																							
links	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0		x				x															x	x	x	x	
1			x					x			x													x	
2							x																		x
3				x	x							x					x								
4	x			x	x							x	x				x	x	x	x					
5				x																					x
6	x				x																x	x	x	x	
7	x				x			x				x	x								x				
8						x																			x
9		x			x	x			x									x							
10				x									x	x	x	x				x					
11		x																	x						
12				x				x													x	x	x	x	
13		x					x			x															
14					x																				x
15	x																					x			
16						x	x			x			x	x	x	x									
17				x			x	x			x									x					
18			x	x				x			x						x	x	x	x				
19	x					x			x					x											
20								x																	x
21	x												x	x	x	x									
22					x	x			x				x	x	x	x		x							
23			x			x	x			x					x								x		
24		x			x										x		x	x	x	x					

Table 6.15: 3444 links - equations mapping (Part I).

	equations																						
links	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	...
0						x																		
1																								
2	x	x	x				x																	
3																								
4								x																
5																								
6									x															
7																								
8	x	x	x							x														
9	x																							
10										x														
11																								
12											x													
13		x																						
14	x	x	x									x												
15																								
16													x											
17			x																					
18														x										
19																								
20	x	x	x												x									
21				x																				
22																x								
23																								
24																	x							

Table 6.16: 3444 links - equations mapping (Part II).

	equations																							
links	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
25	x				x												x	x	x	x					
26			x				x														x	x	x	x	
27			x													x									
28	x				x								x												
29	x				x				x				x												
30	x								x																
31					x				x	x								x							
32						x				x															
33						x				x	x				x										
34											x														
35				x				x				x								x					
36				x								x													
37	x			x	x				x			x	x												
38					x				x																
39	x				x					x								x							
40										x															
41		x				x					x				x										
42				x				x												x					
43			x	x				x			x	x								x					
44								x				x													
45	x			x	x			x	x				x												
46									x																
47	x								x	x								x							
48						x				x					x										

Table 6.17: 3444 links - equations mapping (Part III).

	equations																							
links	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
25					x																			
26																			x					
27																								
28				x	x	x	x													x				
29																				x				
30								x	x	x	x										x			
31																					x			
32												x	x	x	x							x		
33																						x		
34																x	x	x	x				x	
35																							x	
36								x	x	x	x													x
37																								x
38												x	x	x	x									
39																								
40																x	x	x	x					
41																								
42				x	x	x	x																	
43																								
44												x	x	x	x									
45																								
46																x	x	x	x					
47																								
48				x	x	x	x																	

Table 6.18: 3444 links - equations mapping (Part IV).

	equations										
links	49	50	51	52	53	54	55	56	57	58	59
25											
26											
27											
28											
29											
30											
31											
32											
33											
34											
35											
36											
37											
38	x										
39	x										
40		x									
41		x									
42			x								
43			x								
44				x							
45				x							
46					x						
47					x						
48						x					

Table 6.19: 3444 links - equations mapping (Part V).

	equations																							
links	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
49		x								x	x				x										
50			x								x														
51			x	x			x	x				x								x					
52												x													
53	x			x	x			x	x			x	x												
54					x				x									x							
55	x									x								x							
56		x								x															
57		x									x				x										
58							x				x														
59			x	x			x	x			x	x								x					

Table 6.20: 3444 links - equations mapping (Part VI).

	equations																							
links	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49																								
50								x	x	x	x													
51																								
52																x	x	x	x					
53																								
54				x	x	x	x																	
55																								
56								x	x	x	x													
57																								
58												x	x	x	x									
59																								

Table 6.21: 3444 links - equations mapping (Part VII).

	equations										
links	49	50	51	52	53	54	55	56	57	58	59
49						x					
50							x				
51							x				
52								x			
53								x			
54									x		
55									x		
56										x	
57										x	
58											x
59											x

Table 6.22: 3444 links - equations mapping (Part VIII).

6.2 Modeling Large Variants of AES

As can be seen in the previous section of results, the time to solve the various small scale variants increases as the parameters are increased. While Table 6.4 shows the ease of solving $n148$ versions on a single processor, Table 6.14 shows that $n448$ cases are not as quickly solved. In fact, it would probably take longer than the remaining life of the universe to solve $A448$ on a single processor. However, recall that the biclique method of Bogdanov et al. also takes an infeasible amount of time to execute. Therefore, in order to compare our method with others, we create a model to estimate the completion time for the larger variants.

The various tables in this chapter display the total amount of time that the program requires to compute the answer. This includes the initial processing. However, while these milliseconds of initial processing might have a measurable affect on the small *nrce* cases, they are negligible on cases that take hours and/or days to run. In fact, the main consumer of time in the program is the first multi-agree that deletes RHSs. Table 6.23 shows the time results for both multi-agrees that delete RHSs for a sampling of the cases in this research. The second column shows the time for the first multi-agree to execute and the third column shows the time for the second multi-agree to execute (and solve the system). Based on this information, the model attempts to predict the amount of the time required to execute the first multi-agree (as representative of the entire time necessary to solve the system). This is due to the fact that if the first multi-agree is doable, then the second one will also be as it is always significantly faster.

This model for time prediction of the multi-agrees is not a complex model looking for a high degree of accuracy (i.e., to the second). Instead, it focuses on determining how close our method is to breaking AES given current parallel processing resources. This allows a comparison to other algebraic techniques and their results. The model is based on how the program executes multi-agrees. See Appendix A (page 117) for the pseudo-code of the multi-agreeing algorithm. Appendix C (page 151) contains the code of the entire program written in C. Basically, the multi-agree algorithm has three nested loops. The outermost loop figures out the next combination of RHSs to check, the middle loop runs the possible combination through the links in the agreeing order, and the inner loop adds the z values for the combination in the link to determine if it is a valid combination. This yields the model in Equation 6.1:

$$(combos)t_{combo} + (links)t_{link} + (eqn)t_z = \text{total time}. \tag{6.1}$$

nrce	first multi-agree	second multi-agree
3148	1225.75s	28.24s
A418	13585.59s	38.14s
4244	5228.54s	25.01s
7244	13169.94s	10.37s
A244	25898.85s	0.91s
5418	4040.02s	263.25s
A418	15535.45s	655.89s
3424	2992.95s	0.93s
6424	13548.66s	4.51s
A424	36464.41s	0.29s
4228	3331.1s	25.36s
6228	7452.13s	37.84s
A228	19034.63s	23.64s

Table 6.23: Sample of multi-agree times.

The model term *combos* refers to the number of combinations of RHS sets that exist at the start of the multi-agree. Recall this number comes from Equation 5.7. The model term *links* refers to the total number of times that the program will cycle through the link loop during the execution of the multi-agree. The model term *eqn* refers to the number of equations in the link whose values must be added to check if they equal zero (which indicates that the combination of RHSs is valid). The parameter t_{combo} represents the amount of time it takes a combination to be checked. The parameter t_{link} represents the amount of time it takes a link to be found. The parameter t_z represents the amount of time it takes to add a z-value.

Given the speed of current computing resources, t_z is too small to be accurately measured and is, therefore negligible. Therefore, the model was simplified to Equation 6.2.

$$(combos)t_{combo} + (links)t_{link} = \text{total time} \tag{6.2}$$

Equation 6.2 only has two unknowns, yet there exists considerable experimental data to fit to this equation. Therefore, the resulting over-determined system has a relatively small degree of inconsistency between the equations representing each of the AES variants in this research. This is a non-negative least squares problem to solve in which we minimize the norm of $(Ax - b)$ subject to $x \geq 0$. The non-negative feature is important as time cannot be negative with respect

to this problem. Creation of a time machine is not within the scope of this research so only positive time values will be used.

Table 6.24 shows the results of using non-negative least squares techniques in MATLAB to solve for the unknown time values. The model is run using only shape specific data to determine if there was any significant differences based on shape.

Two interesting observations occur. The first observation is that the time values are all in the same order of magnitude. It did not matter that the $n14e$ and $n41e$ cases had 16^4 combinations and the other cases had 16^8 combinations. This lends credence to the validity of the time calculations. Therefore, the last line of Table 6.24 uses all of the experimental data and displays the result. The second observation is that t_{combo} is consistently zero. This indicates that the value of t_{combo} is negligible in finding the total time. In fact, when this model is run without the caveat of allowing only non-negative answers, the value of t_{combo} is always negative.

nrce	t_{combo}	t_{link}
n14e	0	0.000000066532850688
n24e	0	0.000000086488742175
n41e	0	0.000000046250623998
n42e	0	0.000000107006696657
n22e	0	0.000000078661122082
n44e	0	0.000000104071886952
all	0	0.000000099138732423

Table 6.24: Values for AES time unknowns.

The last row of Table 6.24 shows the time values calculated when using all the experimental data in the previous rows. It shows that the time values are fairly consistent over the different sizes and shapes of AES variants. Therefore, the values located in the last row are used in the model calculations.

Recall that the purpose of this model is to predict larger cases of AES that cannot be currently computed. The agreeing order for a specific *nrce* case provides the value of *combos* because this value does not change as field size increases. However, the value of $links$ does change as the field size increases. Therefore, a second model needs to be developed to predict the number of links that a specific case of AES will check.

Links are the total number of iterations of the link loop within the multi-agree algorithm. This

loop checks whether a RHS combination satisfies a link. Each possible combination of RHSs will check between 1 and p links (where p is the number of links being multi-agreed). Therefore, there is an upper and lower bound for the value of links as described in Equation 6.3.

$$(\#\text{ of combinations}) \leq (\#\text{ of links checked}) \leq p * (\#\text{ of combinations}) \tag{6.3}$$

A tighter upper bound is found by examining how the multi-agree algorithm operates. Start with the first combination of RHSs. The algorithm checks all the links in the agreeing order until it reaches a check-for-consistency link. It checks a link by determining whatever the value of each equation's RHS (that is in the link) sums to zero. Because each of these links add at least one new equation to the agreed set, it is trivial to find the RHS for the new equation that keeps the link summed to zero. This changes when the multi-agreeing algorithm encounters a check-for-consistency. Here there are no new equations added to the agreed set, instead there is additional relationship information about the equations already in the set. Therefore, once the algorithm checks the current combination of RHSs against the check-for-consistency link, there is a probability that the RHS set is still valid and the algorithm will continue to the next link. There is also a probability that the set will fail and the algorithm will dump that RHS set and move to the next RHS set to check. The probability that a RHS set will continue through a check-for-consistency is $1/F$, where F is the field size. At each check-for-consistency link, the algorithm is adding all the z values of the RHSs. This summation must equal 0 for the RHS combination to be consistent with the multi-link. Zero is just one value within the field. Since we check each possible combination, it is a good approximation to assume uniform randomness across our sets and that the probability of a set's z sum equaling zero is $1/F$. Therefore, we create a probability tree based on the link ordering to determine how many link checks should occur for any given combination.

Example

This example of the creation of an upper bound for the number of links checked uses the AES case of 3224 and the agreeing order listed in Table 6.12.

The first multi-agree that deletes RHSs occurs with link #13. Figure 6.1 gives a visual representation of the probability tree associated with this case. The first check-for-consistency occurs at link #11 which is the 9th link in the order. Therefore, every RHS combination will check at least nine links. There is a $15/16$ chance that the set will fail and a $1/16$ chance that the multi-agree algorithm will continue. If the algorithm continues, it checks three more links which includes

link #17 which is the next check-for-consistency. There is a $15/16$ chance that the set will fail after checking twelve links in total. There is a $1/16$ chance that the set will be valid and the multi-agree algorithm will continue. In this case, it checks four more links to execute the next check-for-consistency. This is the last link (link #13) for this multi-agree. Regardless of whether the set is valid or not—it will check this last link if the set has lasted this far. This yields a total of $9(\frac{15}{16}) + 12(\frac{15}{16})(\frac{1}{16}) + 16(\frac{1}{16})(\frac{1}{16})(1) = 9.203125$. This number represents the average number of links that will be checked for a RHS combination.

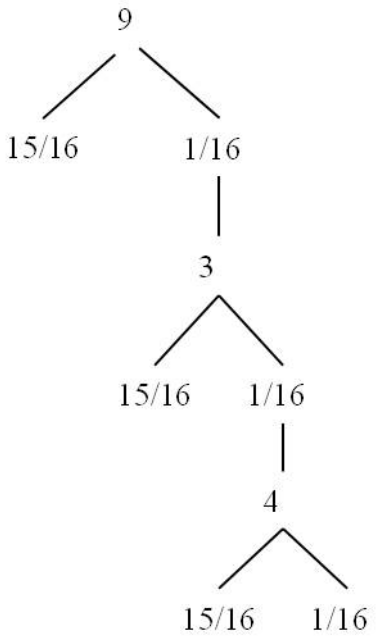

Figure 6.1: Probability tree for 3224.

Multiplying the result of the probability tree by the total number of RHS combinations (in the above example that would be 65,536) should yield the total number of links checked. However, the result from this computation is always larger than the actual number of links checked. This is because the multi-agree algorithm does not always return to the first link in the agreeing order for each new RHS set that is checked. Efficiencies built into the program allow the program to back-track to the first link in which an equation RHS was changed instead of automatically going back to the beginning. Therefore, our model always yields an upper bound to the actual number of links checked.

When executing our model of link counts for the *nrce* variants of AES, it was fairly accurate. Table 6.25 shows the factor that our model is of the actual number of links checked. For ex-

nrce	factor	*nrce*	factor	*nrce*	factor
3412	1.167	3414	1.193	3418	1.200
4412	1.112	4414	1.127	4418	1.130
5412	1.084	5414	1.094	5418	1.097
A412	1.037	A414	1.041	A418	1.042
3142	1.000	3144	1.000	3148	1.000
4142	1.054	4144	1.071	4148	1.077
A142	1.017	A144	1.022	A148	1.023
3222	1.081	3224	1.114	3228	1.124
4222	1.049	4224	1.066	4228	1.071
5222	1.035	5224	1.046	5228	1.050
6222	1.028	6224	1.036	6228	1.038
A222	1.014	A224	1.018	A228	1.020
3422	2.067	3424	2.262		
4422	1.809	4424	1.921		
5422	1.576	5424	1.673		
6422	1.492	6424	1.551		
7422	1.390	7424	1.466		
A422	1.332	A424	1.369		
3242	3.315	3244	4.241		
4242	2.023	4244	2.270		
5242	1.798	5244	1.995		
6242	1.289	6244	1.323		
7242	1.420	7244	1.482		
A242	1.193	A244	1.208		
3442	2.238				
4442	1.438				
5442	1.301				
A442	1.203				

Table 6.25: Factors for link model (organized by shape).

ample, the case of 3224. The algorithm actually checks 541,574 links when it executes the first multi-agree that deletes RHSs. Based on our model of *links*, our model estimates 603,136 links are checked. The factor of 1.114 indicates that our model predicts a number approximately 11% higher that the actual result.

Table 6.25 provides the calculated factors for each case in this research. The results are grouped by shape within each square of the table. Table 6.25 shows that, within a shape, our estimation for the number of links increases its accuracy as the number of rounds increases. It also looks like the accuracy decreases slightly as the field size increases. It clearly shows that our model's

estimation of links checked is fairly close to the actual count. The only significant anomaly is seen in the 324*e* cases where our estimation is three times as large as the actual count. This is because the efficient agreeing order for the other n values in $n24e$ start with a link that has equations from two different intermediate state arrays. The case of $n = 3$ does not have more than one intermediate state array. Therefore, the agreeing order is slightly different which affects our estimation of links. Unfortunately, the agreeing order for 324*e* does not scale up to larger n.

It is also interesting to note that the relative change in factor size from GF(2^4) to GF(2^8) is always less than half of the relative change in factor size from GF(2^2) to GF(2^4). In fact, the change from GF(2^4) to GF(2^8) is between 19% and 50% of the change from GF(2^2) to GF(2^4). The only case that this is not true for is the 314*e* case where it accurately estimates (to three significant digits) the number of links checked. Therefore, an upper bound exists that the jump in relative change in factor size from GF(2^4) to GF(2^8) is 50%. Using that figure, factors are estimated for $n428$ and $n248$ cases. The results are shown in Table 6.26.

nrce	factor
3428	2.369
4428	1.980
5428	1.724
6428	1.582
7428	1.506
A428	1.388
3248	4.833
4248	2.409
5248	2.104
6248	1.340
7248	1.514
A248	1.216

Table 6.26: Estimated factors for $n248$ and $n428$ AES variants.

This clearly shows that our estimate for the number of link loops executed gets closer to the correct number as the size of the MRHS system grows. Therefore, a factor of 1.00 (namely, using our estimated number of link loops as the value for *links*) is an absolute upper bound for our time calculations. This is the factor that is used in the estimation of times. The results are shown in Table 6.27. Then, using the time estimates from Table 6.24 and the model in Equation 6.2, an upper bound is calculated on the time needed to solve these cases.

Estimating factors for cases of $n44e$ is slightly more difficult as experimental data is only avail-

nrce	factor	estimated time
3428	1.00	1,333,321 years
4428	1.00	1,796,702 years
5428	1.00	2,260,761 years
6428	1.00	3,071,849 years
7428	1.00	3,709,305 years
A428	1.00	5,737,807 years
3248	1.00	1,043,794 years
4248	1.00	1,912,602 years
5248	1.00	2,028,506 years
6248	1.00	2,318,487 years
7248	1.00	3,303,421 years
A248	1.00	4,636,517 years

Table 6.27: Estimated solution times for $n248$ and $n428$ AES variants.

able for the 4-element field. Examining the data collected from each of the AES variants shows that the largest relative increase in factor for any variant (excluding the anomaly of $324e$) was a 12.2% increase. Therefore, we can assume that this percent does not increase and use it to estimate the factors for $n444$ cases. The same methodology for computing the factors in Table 6.26 is used to compute the factors for $n448$. These results are shown in Table 6.28.

nrce	factor
3444	2.511
4444	1.613
5444	1.460
A444	1.350
3448	2.664
4448	1.712
5448	1.549
A448	1.432

Table 6.28: Estimated factors for $n444$ and $n448$ AES variants.

However, ultimately a factor of 1.00 is used to calculate the number of link loops to be executed in the algorithm. This ensures that the resulting time estimate is an upper bound to what is required. The results are shown in Table 6.29.

See Section 8.2 for a discussion on how extensions to this research may suitably reduce these times.

nrce	factor	estimated time
3444	1.00	$1,626,727$ years
4444	1.00	$3,075,269$ years
5444	1.00	$3,253,213$ years
A444	1.00	$9,048,501$ years
3448	1.00	$2.994 \cdot 10^{25}$ years
4448	1.00	$5.666 \cdot 10^{25}$ years
5448	1.00	$5.987 \cdot 10^{25}$ years
A448	1.00	$1.668 \cdot 10^{26}$ years

Table 6.29: Upper bounds for solution times for *n*444 and *n*448 AES variants.

6.3 Other AES Results

These extensions of the original MRHS concept are not the only ideas being applied to AES. A few other researchers have published their timing results on small scale AES variants. Many of these researchers demonstrate the viability of their method on smaller variants than the ones used in this research. This helps to show that our method is very competitive.

Thorsten Schilling

Thorsten Schilling [29] extends the ideas of Raddum and Semaev with the concept of learning. Learning is a guess and check algorithm. It is designed to verify the correctness of a variable guess quickly and then easily backtrack through the symbols as necessary. During the search for the solution, the learning algorithm obtains new information from the wrong guesses (namely, 'why' the guess was wrong and which symbols caused it to fail) and applies that knowledge when determining the next guess to make. Thorsten compares his results to the MiniSat technique.

Kenneth Matheis

In [30], Kenneth Matheis proposes a hardware design for implementing MRHS that has significant performance gains compared to implementing Raddum and Semaev's original work in software. The total chip area of the design is 2.25 m^2 and it has a storage capacity of 4.792 TB. For the small scale AES variant of 3448, Matheis estimates that it would take $5.726 \cdot 10^{23}$ years years to solve while guessing 108 bits out of the 128 key bits. This is faster than our methods on a single processor. However, if we "guessed" 108 bits of the key, that would eliminate most of the RHS's and our algorithm would become orders of magnitude faster.

Sean Simmons

Sean applies the F4 algorithm to small scale variants of AES in his paper [31]. His focus was the AES small scale variant of 2224. In the 2224 case, he uses 8 plaintext/ciphertext pairs to determine the solution in 106 - 329 seconds on a 2.4 GHz processor. While we ignore the $n = 2$ cases, our 3224 case uses 1 plaintext/ciphertext pair and finds the solution in 0.030 seconds using a slower processor. He was able to reduce his computational time on 2224 by using 16 plaintext/ciphertext pairs. However, the time merely reduces to a time of 34 - 38 seconds. It is clear that our method is more efficient in determining the solution.

Elizabeth Kleiman

Her PhD dissertation [32] examines Baby Rijndael ($n224$) and parallel processing possibilities to solve it. Her results indicate that XL and XSL can't handle the size of the equations for 4224. Our methods found the solution to 4224 in 0.054 seconds. Additionally, she hypothesized that XSL on 4224 would work with two plaintext/ciphertext pairs but she ran out of computing power when conducting her research.

Stanislav Bulygin and Michael Brickenstein

In 2008, these gentlemen wrote [33] about constructing a zero-dimensional Gröbner representation for AES in GF(2). They use the PolyBoRi [34] algorithm on an AMD 2.2 GHz processor to solve A224 in 1205 seconds. Our A224 case took 0.291 seconds to solve. They also used PolyBoRi to conduct a meet-in-the-middle attack on AES variants using numerous pairs of plaintext/ciphertext. These meet-in-the-middle attacks were conducted on $n = 2$ variants of AES. As we did not analyze these variants in this research because they were non-conducive to scaling, a direct comparison cannot be made at this time. Finally, they explore how guessing bits of the key can help obtain the solution to the system quicker. For the case of 3244—they used 256 pairs of plaintext/ciphertext and guessed seven bits (approximately 21% of the key). They were able to solve this in 657 seconds. Our method (with 1 plaintext/ciphertext pair and no guessing) took 800 seconds.

6.4 Multiple Plaintext/Ciphertext Pairs

The results discussed in this chapter are based on using one plaintext/ciphertext pair of data to create the equations. Many current cryptanalysis methods require multiple pairs of plaintext/-ciphertext to operate. For example, the "Lucky 13" attack [35] is able to recover the plaintext of website authentication cookies with only 2^{13} plaintext/ciphertext pairs of data. Another example is the latest attack on RC4 that requires one billion (2^{30}) plaintext/ciphertext pairs to find the key. While seeming like a large number, it takes only 32 hours to currently execute [36].

The use of multiple plaintext/ciphertext pairs is a common input to breaking cryptosystems because this additional data tends to simplify the problem. In general, the more information one has about a problem—the easier it is to solve. Therefore, part of this research focuses on multi-agreeing orders that use information from multiple plaintext/ciphertext pairs created from the same key.

MRHS equations that are generated from multiple plaintext/ciphertext pairs (that are encrypted with the same key) are slightly different than the equations generated from only one plaintext/ciphertext pair. Namely, there are more variables, which results in more equations, and therefore, more links are created. However, since the key is the same—the key schedule equations and key variables are the same. Assume you are using m plaintext/ciphertext pairs. Now, instead of $nr + nrc = nr(c+1)$ equations there are $nr + m(nrc) = nr(mc+1)$ equations. Originally, there were $rc + nr + (n-1)rc$ variables. Now, there are $rc + nr + m(n-1)rc$ variables. Let $k = nr(mc+1)$ which represents the number of equations. Let j represent the number of variables. Therefore, $j = rc + nr + (n-1)rc = nr(mc+1) + rc(1-m) = k + rc(1-m)$. Since $m \geq 2$, $j < k$.

Recall the example of 3224 in Table 6.12 and Figure 6.1. This variant has eighteen equations. The agreeing order has the 16th link delete RHSs and has a maximum of four degrees of freedom during the process. Let $m = 3$. Now there are 42 equations. For multiple plaintext/ciphertext to be helpful in solving the system, it needs to delete RHSs quicker than the single case. Based on Equation 6.2, there are two options to reduce the total time. The first is to create an ordering which has a multi-agree that deletes RHS earlier than the 16th link (which translates to using, at most, (15/42) of total information about the system instead of using (16/18) of the information on the system). The second is to reduce the number of free equations within the order to shrink the number of consistent RHS sets. Therefore, viable strategies must create one of these opportunities. Three multi-agreeing concepts are explored in this research.

Strategy #1.

First, I attempt to map the agreeing order (for a specific *nrce* variant of AES) from one plaintext/ciphertext pair to m plaintext/ciphertexts. For instance, assume Link #5 relates a key variable to the first state array (in the first pair) and Link #29 relates the same key variable to the first state array (in the second pair). These parallels in link structure exist in the default set of links due to their systematic method of construction. Therefore, perhaps agreeing Link #29 after Link #5 in the agreeing order would be helpful as it would provide additional information about the

key variable in question.

Example.
Consider the case of 3224 using two plaintext/ciphertext pairs. Variables #0 to #9 are key variables, Variables #10 to #17 are intermediate state array variables from the first plaintext/ciphertext pair, and Variables #18 to #25 are intermediate state array variables from the second plaintext/ciphertext pair. Creating the default set of links yields Links #13 and #21 that exclusively deal with the key schedule equations (which are the same for both pairs). Table 6.30 gives the mapping of the remainder of the links between the two plaintext/ciphertext pairs. By 'mapping', we mean that the links relate the same relative variables and equations together for each plaintext/ciphertext pair.

1st pt/ct	2nd pt/ct
Link #0	Link #24
Link #1	Link #10
Link #2	Link #26
Link #3	Link #27
Link #4	Link #28
Link #5	Link #29
Link #6	Link #30
Link #7	Link #31
Link #8	Link #32
Link #9	Link #33
Link #11	Link #16
Link #15	Link #18
Link #17	Link #12
Link #19	Link #20
Link #23	Link #22
Link #25	Link #14

Table 6.30: Mapping of multiple pt/ct links.

Then the original single agree order (see Table 6.12) is applied to the problem. First, this order is applied to just the information from the first plaintext/ciphertext pair in this example. The first multi-agree that deletes RHSs occurs on the 16th link with four degrees of freedom and took 0.085 seconds to execute. Second, this order is applied to just the information from the second plaintext/ciphertext in this example. The first multi-agree that deletes RHSs occurs on the 16th link with four degrees of freedom and took 0.086 seconds to execute. Third, this order is applied to using both orders, one right after another. Due to the fact that there are more links

link number	new eqns to agreed set	# degrees of freedom	comment
Link #11	0, 1, 2, 9	3	
Link #16	21	3	
Link #1	3, 6	4	
Link #10	18	4	
Link #17	7	4	
Link #12	19	4	
Link #25	8	4	
Link #14	20	4	
Link #15	10	4	
Link #18	22	4	
Link #19	11	4	
Link #20	23	4	
Link #21	4	4	key equations only
Link #2	14	4	
Link #3	–	3	check-for-consistency
Link #26	26	3	
Link #27	–	2	check-for-consistency
Link #13	5	2	key equations only
Link #8	17	2	
Link #9	–	1	check-for-consistency
Link #32	29	1	
Link #33	–	0	solved

Table 6.31: Agreeing order for two pairs of 3224.

than variables—not all of the links are needed in the multi-agreeing order in order to solve the system. However, RHSs are not deleted in this order until the 22nd link. Table 6.31 shows this multi-agreeing order utilizing two plaintext/ciphertext pairs.

However, this method fails to improve the use of a single plaintext/ciphertext pair. Recall that the resulting time from Equation 6.2 is decreased by either shrinking the number of RHS combinations (by decreasing the total degrees of freedom within the order) or by forcing the first multi-agree to occur earlier in the order. This method maintains the same number of degrees of freedom as the single plaintext/ciphertext order because it is based on the same idea. It also fails to force a check-for-consistency earlier in the order. This is because the multi-agrees that delete RHS sets are happening at the same relative place in the order. Yet, using the links from both plaintext/ciphertext pairs causes the multi-agrees to be triggered after more links are in the agreeing order (in this example they occurred after the 22nd link instead of the 16th link). This also increases the number of equations in the system that must be checked in the links.

Specifically, the the order in Table 6.31 took 0.130 seconds to execute.

Clearly, applying this strategy to multiple plaintext/ciphtertext pairs takes longer than just using one plaintext/ciphertext pair. In addition, it is found in most experimental cases, that even if the first multi-agree in the order deletes RHSs (which it did not), it still takes longer using multiple plaintext/ciphertext pairs than just using one to find a solution to the system. Ultimately, the problem is that additional plaintext/ciphertext pairs are just complicating the situation.

Strategy #2.

With Strategy #1 not displaying promising results, I discarded the original multi-agreeing orders and attempted to find an unique ordering for cases of multiple plaintext/ciphertext pairs. Unfortunately, nothing was consistently found that would cause multi-agrees to trigger earlier and/or with a smaller sets of RHSs. Similar to the single plaintext/ciphertext case, some efficient orders are found for the abnormal cases (i.e., $n = 2$ and $e = 2$), but nothing is able to extend to other sizes. For example, an agreeing order is found for the 3224 example used in Strategy #1. It allows no excessive links to be included into the order and focuses on the early equations. It took 0.035 seconds to execute. The resulting agreeing order is displayed in Table 6.32.

However, this example is an abnormality, as it could not be scaled to larger values of n. Perhaps the method of MRHS using an independent set of links is just not conducive to utilizing more than one plaintext/ciphertext pair.

Strategy #3.

The last concept explored in this research was the use of dependent links to assist the multi-agreeing order. Dependent links do not increase the amount of information about the system. Therefore, they cause the number of links before a multi-agree to increase. However, if they can reduce the number of degrees of freedom in the multi-agreeing order—this is good. The reduction of the exponential portion of the timing model is more helpful (and could overcome) an increase in the linear portion.

At this state of the research, the creation of dependent links are extremely time consuming, since they are crafted by hand. The concept is to take the 'best' multi-agreeing order that is constructed with the default set of multi-links. Then one examines every time an increase in the degrees of freedom is introduced in the order. The focus is to then create a custom link (or a series of custom links) that introduces these new equations into the agreed set without

link number	new eqns to agreed set	# degrees of freedom	comment
Link #11	0, 1, 2, 9	3	
Link #13	5	3	
Link #17	7	3	
Link #25	8	3	
Link #19	11	3	
Link #23	3, 12	4	
Link #1	6	4	
Link #15	10	4	
Link #9	17	4	
Link #8	–	3	check-for-consistency
Link #0	13	3	
Link #5	15	3	
Link #4	–	2	check-for-consistency
Link #21	4	2	
Link #7	16	2	
Link #6	–	1	check-for-consistency
Link #3	14	1	
Link #2	–	0	solved

Table 6.32: More efficient agreeing order for 2 pairs of 3224.

increasing the number of consistent RHS sets. In general, this tends to add 2-3 multi-links to replace the increase in degrees of freedom. This method does not work on every increase in the number of degrees of freedom nor on every variant. While this research did occasionally find a series of custom multi-links that could replace increases of degrees of freedom, it was only in abnormal, small variants.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 7:
Other Cryptosystems

Prior to Daemen and Rijmen's creation of the well-known Rijndael, they created other non-Fiestel block ciphers. These are Shark (1996) and Square (1997). These iterative ciphers were created in accordance with the wide-trail design strategy, and inherited useful properties from their predecessors. Due to their structural similarity to AES, it is possible to create a MRHS system of equations to describe these ciphers and, therefore, utilize our previously described techniques on these representations. A description of the ciphers (to include their differences from AES), insights gained from the technique execution, and the results are located in this chapter. Additionally, the cipher Anubis is discussed. Rijmen created Anubis, a descendant of AES, in 2000.

7.1 Square

Square is a 128 bit block cipher revealed to the world in 1997 by Joan Daemen, Lars Knudsen, and Vincent Rijmen. It is the direct predecessor (parent) of AES. It builds off the successes of Shark and refines them. Square's primary design goal is to scale efficiently on a range of processors and run in parallel within hardware.

Like AES, Square consists of identical rounds of transformations. The initial cipher release consisted of six rounds, but analysts found a dedicated, chosen-plaintext attack on this size. The final cipher version was changed to eight rounds. Each round has four transformations that, like AES, can be implemented as XORs and table lookups [37].

The four Square transformations are linear, non-linear, permutation, and key addition. In AES, the linear transformation is the Mix Columns step which multiplies the state array by a particular circulant matrix to mix the columns. In Square's linear transformation, an upper triangular matrix multiplies the state array. This multiplication operates on the rows of the state array instead of the columns (as in AES). According to its designers, the particular upper triangular matrix was selected to maximize the branch number and facilitate implementation on 8-bit processors [18, p.166].

The non-linear transformation in Square is a Byte Substitution operation using a specially crafted S-box. This is the same concept used in AES. In AES, the S-box consists of the in-

version operation (on bytes) followed by a specific affine transformation (see page 16). In Square, the S-box also uses inversion but with a different affine transformation. The Square affine transformation is over GF(2) and has a complicated description in GF(2^8). However, a complicated description in GF(2^8) does not increase the complexity of the MRHS equations. The same number of right-hand sides $\overrightarrow{b}$ exist for MRHS equations in Square as they did in AES.

In AES, the permutation step is the Shift Rows operation. In Square, the permutation occurs when the rows of the State Array are transposed with its columns. This involution is nice and compact for implementation. However, it does create differences between the MRHS representations of small scale variants of Square versus AES. Shift Rows in AES can be easily implemented on any shape. It is especially useful to analyze $n14e$ AES, which negates Shift Rows so one can see the interaction of the other transformations. However, in rectangular Square variants, the concept of interchanging rows and columns becomes tricky. The result is that there are slight differences in efficient agreeing orders for odd n and even n Square variants.

The final round element of Square is the Key Addition transformation. The add key step is the exact same for both AES and Square; it is the bitwise addition of the round key.

In both ciphers' structure, there is a requirement for a different round key for each round of the algorithm. However, these two algorithms use vastly different implementations. In AES, the key schedule algorithm is non-linear and uses the same strong S-boxes as the main encryption algorithm. In Square, the key schedule algorithm is linear and implemented as an affine transformation. The key schedule seems complicated in its design. It consists of multiple bytes shifting positions within each row and then either adding a constant or a previous row value to each byte. The Square key schedule is implemented using an iterative definition. Due to the linear nature of this component, the final algorithm should be easier to solve because a level of complexity has been removed from the process. The trade-off is that more variables exist within Square's MRHS representation than AES, and these variables are more prevalent in each equation to handle the key relationships.

7.1.1 Implementation

Due to Square's many similarities with AES, it is relatively straightforward to create the MRHS equations that represent Square. See Appendix E for the C code that creates the Square MRHS equations. Many key insights are gained from looking at smaller versions of AES before ana-

lyzing the full version and therefore, Square is tackled similarly. This research analyzes mini-Square (2x2) and full-Square (4x4) with a varying number of rounds. Even though the name of the cipher is Square, the cipher's implementation is modified so that rectangular shapes are also generated and analyzed. The sample implementation [38] given by the authors and their published paper [37] assist in our creation of Square's MRHS representation.

It is a huge jump to move from mini-Square to full-Square in terms of the complexity of designing a multi-agreeing order. Therefore, rectangular shapes of state arrays are examined for insights. Having a rectangular matrix complicates the implementation of the Transpose transformation. Therefore, the dimensions of the state array are changed multiple times during the process (i.e., $r \times c$ becomes $c \times r$). This results in a process that treats rows and columns in the same way with regards to the Transpose transformation. Therefore, there are no differences between cases of $n41e$ and $n14e$ nor between cases of $n42e$ and $n24e$. It is the number of variables in Square's state array that is helpful to look at instead of a specific rectangular shape.

Although the *nrce* definition is created for AES, I will also use it to describe Square cases as the reader should be comfortable with the meaning of this notation.

7.1.2 Notation

A typical process of Square looks like

$$M^{-1}A \quad MSTA \quad MSTA \quad \ldots \quad MSTA \quad MSTA,$$

with letters standing for the following transformations described in this section: M (Multiplication by a upper-triangular matrix), S (Byte Substitution with S-box), T (Transpose—the permutation transformation), and A (Add Round key). Similar to AES, these transformations are re-grouped without changing the results. This creates the following process:

$$M^{-1}AMST \quad AMST \quad \ldots \quad AMST \quad AMST \quad A.$$

The following notation is used to create the equations: n is the number of rounds, P is plaintext, C is ciphertext, I is the intermediate state array after a round key is added, and X is the state array at the end of the byte substitution (and transpose) operation. Then the process looks like

(with output [states]):

$$[P]\,M^{-1}\,AM\,[I_0]\,ST\,[X_1] \quad AM\,[I_1]\,ST\,[X_2] \quad \ldots \quad AM\,[I_{n-2}]\,ST\,[X_{n-1}] \quad AM\,[I_{n-1}]\,ST\,[X_n] \quad A\,[C].$$

This process yields the following algebraic equations:

$$I_0 = P + M(K_0) \quad T(X_i) = S(I_{i-1}) \quad I_i = M(X_i + K_i) \quad C = X_n + K_n\ . \tag{7.1}$$

These translate into the following MRHS equations:

$$\begin{aligned}
\begin{bmatrix} (M(K_0))_{j,k} \\ X_{1,k,j} \end{bmatrix} &= \begin{bmatrix} S_{\text{in}} \\ S_{\text{out}} \end{bmatrix} + \begin{bmatrix} P_{j,k} \\ 0 \end{bmatrix}; \\
\begin{bmatrix} I_{i,j,k} \\ X_{i+1,k,j} \end{bmatrix} &= \begin{bmatrix} S_{\text{in}} \\ S_{\text{out}} \end{bmatrix} \quad 1 \le i \le n-2; \\
\begin{bmatrix} I_{n-1,j,k} \\ K_{n,k,j} \end{bmatrix} &= \begin{bmatrix} S_{\text{in}} \\ S_{\text{out}} \end{bmatrix} + \begin{bmatrix} 0 \\ C_{k,j} \end{bmatrix}.
\end{aligned} \tag{7.2}$$

The key schedule algorithm is similiar to AES but without the S-box substitution. It is written in terms of columns:

$$K_{i,0} = F(K_{i-1,c-1}) + (\text{if } c > 1)\ K_{i-1,0} \quad K_{i,j} = K_{i-1,j} + K_{i,j-1} \quad i, j > 0,$$

where row k of the output of the column rotation F is

$$F_k(K_{i,j}) = K_{i,j,(k+1 \mod r)} + (\text{if } k = 0)\ 2^i.$$

This gives no MRHS equations, since the key schedule contains no non-linearity.

The ordering of the variables puts the key variables $K_{0,j,k}$ before those for states X_i. Additionally, the intermediate states I_i and non-initial round keys K_i get replaced by equivalents in terms of actual variables.

7.1.3 Application of Technique

Using a similar process as with AES, efficient agreeing orders are found for $n14e$, $n42e$, $n22e$ and $n44e$. I use many of the insights gained from analyzing AES. In general, the Square multi-agreeing orders are easier to find than for AES as there are fewer possibilities that keep the degrees of freedom minimized. All Square results are computed on a single 2.4 GHz Intel processor with 4 GB of memory. See Appendix F for the C code that implements our methods on Square.

$n14e$

Unlike AES, Square uses only eight rounds of transformations. Therefore, this research analyzed $214e$, $314e$, $414e$, $514e$, and $814e$. However, like AES, $n = 2$ and $n = 3$ were not helpful cases to analyze. This is because the patterns for multi-agreeing orders in these cases did not scale up to the larger round variants as the equations and their links directly map plaintext to ciphertext. Agreeing orders (for $n = 2$ and $n = 3$) were found that only used three free equations. Unfortunately, this did not scale up to larger rounds.

The best agreeing order for this size of state array contained four free equations. Dependent link creation was also attempted to assist with this case. Unfortunately, dependent links did not help reduce the time required to compute the solution to the system. The final multi-agreeing order for $n14e$ started with the link containing Equations #0 through #4. Equations #0 through #3 are the most prolific equations in all the links. Recall that Equations #0 through #3 had the plaintext incorporated within them. (See Table 7.1 for a visualization of this observation.) In Table 7.1, Equations #12 through #15 had the ciphertext incorporated within them.

The multi-agreeing order for 4144 begins with Link #7. After this link, the remainder of the links are placed in the agreeing order in such a way that the remaining equations are agreed into the set in numerical order. Continuing with the example of the 4144 case, the agreeing order follows with Link #3 (introducing Equation #5), Link #1 (introducing Equation #6), Link #5 (introducing Equation #7), Link #0 (introducing Equation #8), Link #2 (introducing Equation #9), Link #4 (introducing Equation #10), Link #6 (introducing Equation #11), Link #9 (introducing Equation #12), Link #8 (check-for-consistency), Link #11 (introducing Equation #13), Link #10 (check-for-consistency), Link #13 (introducing Equation #14), Link #12 (check-for-consistency), Link #15 (introducing Equation #15), and Link #14 (check-for-consistency to solve).

	equations															
links	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	x	x			x				x							
1	x	x	x	x			x									
2	x	x	x			x				x						
3	x	x	x	x		x										
4	x	x	x	x			x				x					
5	x	x	x					x								
6		x	x	x				x				x				
7	x	x	x	x	x											
8		x		x					x	x	x	x	x			
9			x										x			
10	x	x		x					x	x	x	x		x		
11	x	x		x										x		
12	x	x	x	x					x	x	x	x			x	
13	x		x												x	
14		x	x						x	x	x	x				x
15		x	x													x

Table 7.1: Square 4144 links - equations mapping.

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
4142	4	0.002s	4144	3	0.027s	4148	2	25m 33s
5142	1	0.000s	5144	3	0.042s	5148	3	44m 2s
8142	1	0.002s	8144	1	0.080s	8148	2	1h 25m 57s

Table 7.2: Square $n14e$ MRHS representation stats.

Both $n14e$ and $n41e$ Square cases result in four degrees of freedom in their multi-agreeing order. Therefore, there are four multi-agrees that occur with check-for-consistency links. However, only the last two multi-agrees delete RHSs in the case of $e = 4$ and $e = 8$. The results of $n14e$ Square are located in Table 7.2.

*n*42*e*

The next size of a Square state array is analyzed in the $n42e$ cases. Similar to the $n14e$ case, the resulting multi-links are quite dense with equations. Virtually all of the links use an average of 90% of the first eight equations within them. These equations are the ones that have the plaintext incorporated within them. Additionally, the last sixteen links have the last eight equations in them. These equations have the ciphertext incorporated in them. These links result in checks for consistency within our ordering method. The density of equations in these multi-links (in comparison to AES multi-links) force a large number of degrees of freedom within the multi-

	equations															
links	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	x															
1																
2		x														
3																
4			x													
5																
6				x												
16	x	x							x							
17									x							
18			x	x						x						
19										x						
24	x	x											x			
25													x			
26			x	x										x		
27														x		

Table 7.3: Portion of the Square 4424 links - equations mapping.

agreeing order. From this perspective, AES was nicer to analyze. This is because there are more initial possibilities for agreeing orders based on the default set of links. There are only limited possibilities for the Square cases. Therefore, using custom links and changing the basis might work well with Square in future research.

Different ordering schemes were attempted on this Square case to reduce the maximum degrees of freedom. Ultimately, an ordering similar in concept to $n14e$ was used. The link consisting of Equations #0 through #9 was the starting link. Each subsequent link introduced the remaining equations in numerical order. However, a slight difference in the location of the checks for consistency links within the multi-agreeing order occurred depending on whether the case had an odd or an even number of rounds. Table 7.3 shows a portion of the mapping of equations to links for 4424. Notice that each of the displayed links has a pair of equations from the second to last state array within them along with a final state equation. While not captured in this portion of the table, these links also have multiple earlier equations within them. Assume the agreeing order has brought Equations #0 through #15 into the set. The order would continue as follows: Link #0 (introducing Equation #16), Link #2 (introducing Equation #17), Link #17 (introducing Equation #24), Link #16 (check-for-consistency), then Link #25 (introducing Equation #28), and Link #24 (check-for-consistency). This pattern then repeats with all the pairs—two state

	equations															
links	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
8	x															
9																
10		x														
11																
12			x													
13																
14				x												
24	x	x	x	x					x							
25									x							
26					x	x	x	x		x						
27										x						
28	x	x	x	x							x					
29											x					
32	x	x	x	x									x			
33													x			
36	x	x	x	x											x	
37															x	

Table 7.4: Portion of the Square 5424 links - equations mapping.

equations are introduced, then two sets of checks for consistency are executed. This means that the first check-for-consistency link for 4224 occurs at the 31st link out of 48 total links.

Table 7.4 shows a portion of the mapping of equations to links for 5424. In this odd-n case, there are now quads of sequential equations within these links. Therefore, we will have to bring in all four of these equations separately first before including the check-for-consistency links. This means that the odd n cases will take longer to multi-agree because check-for-consistency links happen later within the order. Assume the agreeing order has brought Equations #0 through #23 into the set. The order would continue as follows: Link #8 (introducing Equation #24), Link #10 (introducing Equation #25), Link #12 (introducing Equation #26), Link #14 (introducing Equation #27), Link #25 (introducing Equation #32), Link #24 (check-for-consistency), then Link #29 (introducing Equation #34), Link #28 (check-for-consistency), then Link #33 (introducing Equation #36), Link #32 (check-for-consistency), then Link #37 (introducing Equation #38), and Link #36 (check-for-consistency). This pattern then repeats once more with the second set of sequential quads. This means that the first check-for-consistency link for 5424 occurs at 39th link out of 60 links.

The results of *n*42*e* Square are located in Table 7.5.

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
3422	1	0.022s	3424	3	35m 1s	3428	⋆	⋆
4422	1	0.043s	4424	3	1h 5m 50s	4428	⋆	⋆
5422	1	0.092s	5424	3	2h 4m 13s	5428	⋆	⋆
8422	1	0.195s	8424	1	4h 12m 24s	8428	⋆	⋆

⋆ No solution found within 300 hours of computation time.

Table 7.5: Square *n*42*e* MRHS representation stats.

*n*22*e*

	equations															
links	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	x	x	x		x	x			x							
1	x	x	x				x									
2			x	x			x	x		x						
3	x	x	x	x		x										
4	x		x	x	x	x					x					
5		x	x	x				x								
6	x	x					x	x				x				
7	x	x		x	x											
8			x						x	x			x			
9	x	x	x	x									x			
10				x							x	x		x		
11	x	x	x	x										x		
12	x								x	x					x	
13	x	x	x	x											x	
14		x									x	x				x
15	x	x	x	x												x

Table 7.6: Square 4224 links - equations mapping.

All links in the *n*22*e* cases of Square have a multitude of equations within them like previous Square variants. Therefore, we are faced with using the same initially expensive links to start the process of multi-agreeing. When working with mini-Square, it was determined that the fastest ordering had a starting link of five total equations (four degrees of freedom) and then the order decreased the degrees of freedom with check-for-consistency links. The most efficient starting link was the one that used the four most prolific equations and the last state array equation.

This concept of starting link selection is slightly different than what worked for the rectangular shaped Square variants. Table 7.6 shows the mapping of links to equations for 4224. The most efficient starting link was found to be Link #15.

Notice the identical structures of Links #9, #11, #13, and #15. They all use the first four equations (the ones that incorporate the plaintext) plus one of the last equations (the ones that incorporate the ciphertext). Testing showed that it was more efficient to first agree these links in to the order and then use their counterparts (Links #8, #10, #12, and #14, respectively) as the check-for-consistency. Table 7.7 gives the agreeing order for 4224.

link number	new eqns to agreed set	# degrees of freedom	comment
Link #15	0,1,2,3,15	4	
Link #7	4	4	
Link #3	5	4	
Link #1	6	4	
Link #5	7	4	
Link #0	8	4	
Link #2	9	4	
Link #8	12	4	
Link #9	–	3	check-for-consistency
Link #12	14	3	
Link #13	–	2	check-for-consistency
Link #4	10	2	
Link #6	11	2	
Link #14	–	1	check-for-consistency
Link #10	13	1	
Link #11	–	0	solution

Table 7.7: Agreeing order for Square 4224.

The results of *n*22*e* Square are located in Table 7.8.

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
3222	4	0.002s	3224	1	0.006s	3228	2	4m 58s
4222	1	0.000s	4224	1	0.021s	4228	1	19m 11s
5222	4	0.001s	5224	2	0.055s	5228	3	59m 50s
8222	1	0.001s	8224	1	0.108s	8228	3	1h 5m 55s

Table 7.8: Square *n*22*e* MRHS representation stats.

*n*44*e*

The Square variants of *n*44*e* are very similar in structure to the previously described cases. For instance, Equations #0 through #15 (the equations with plaintext incorporated into them), are heavily present in each link. The *n*44*e* links use, on average, 8-9 of these equations. Additionally, for the cases of $n = 2$ and $n = 3$, multi-agree orderings were found that used less than sixteen degrees of freedom. Unfortunately, this was not possible with $n \geq 4$. Also, as was true with previous variants, only the last two multi-agrees actually delete RHSs.

The most efficient multi-agree ordering for *n*44*e* brought the equations into the agreeing order in numerical order. Namely, the first link should include Equation #16, the next link should include Equation #17, and so forth. Once the first four links are in the multi-agreeing order, this ensures that all the initial Equations are in (i.e., Equations #0 through #15) and the first quad of sequential state array equations are in. This process continues until the first check-for-consistency link is available. These check-for-consistency links exist as the last 32 multi-links when using the default link creation method and also occur in quads. Namely, the ordering will execute four checks for consistency (using a pair of multi-links), then it will bring in four additional links, then four checks for consistency, then four additional links, and so forth until the last multi-link is used.

Experimentation was performed to determine if there were more efficient ways to introduce the prolific equations into the multi-agreeing order. However, these attempts only served to increase the degrees of freedom in the method. Therefore, they were discarded. Attempts were also made to introduce the check-for-consistency links earlier within the multi-agreeing order. However, these also failed to find a solution faster.

The results of *n*44*e* Square are located in Table 7.9.

nrce	# soln's	time	*nrce*	# soln's	time	*nrce*	# soln's	time
2442	1	23.8s	2444	$\star$	$\star$	2448	$\star$	$\star$
3442	1	33m 25s	3444	$\star$	$\star$	3448	$\star$	$\star$
4442	4	2h 7m 39s	4444	$\star$	$\star$	4448	$\star$	$\star$
5442	4	6h 14m 21s	5444	$\star$	$\star$	5448	$\star$	$\star$
8442	1	8h 0m 45s	8444	$\star$	$\star$	8448	$\star$	$\star$

$\star$ No solution found within 300 hours of computation time.

Table 7.9: Square *n*44*e* MRHS representation stats.

7.1.4 Modeling larger cases

Like AES, many of the larger variants of Square require an infeasible amount of time to run on a single processor. Using the same model construction methodology as AES (Page 81), a model is created to estimate the time completion of the larger variants of Square. Equation 6.2 is used and Table 7.10 displays the MATLAB results of using non-negative least squares techniques on the Square experimental data. The time values calculated are the same order of magnitude as was found for AES.

nrce	t_{combo}	t_{link}
n14e	0	0.000000043483344263
n42e	0	0.000000079024947453
n22e	0	0.000000051527389281
n44e	0.000000055166130583	0.000000079100354931
all	0	0.000000076387648850

Table 7.10: Values for Square model unknowns.

The number of iterations of the link loop are also calculated in the same manner as for AES. Recall Figure 6.1. The results are also strictly upper bounds for the actual number of link loops executed in the multi-agree program. Table 7.11 displays the factors for the Square variants. The first observation is that our prediction for the number of iterations of the link loop for Square is closer to the actual results than AES. There is also the out-lying case of 8222 where our model predicts a smaller number of link loops than the program is forced to execute during the multi-agree function.

Using a factor of 1.0 for the link loops and then the time values calculated in Table 7.10, completion times are estimated for the larger variants of Square. The results are located in Table 7.12.

7.2 Shark

Shark (1996) is a joint creation by Vincent Rijmen, Joan Daemen, Bart Preneel, Antoon Bosselaers, and Erik DeWin. Like Square, Shark is a generational parent of AES. The stated purpose of this cipher [39] is to create a fast software implementation cipher which is resistant to differential and linear cryptanalysis. Shark's six rounds consist of only two transformations (unlike the four in AES and Square). These two transformations are a non-linear transformation and a diffusion transformation. Shark also uses a key schedule algorithm like its successors. Shark's

nrce	factor	nrce	factor	nrce	factor
4142	1.000	4144	0.999	4148	1.000
5142	1.000	5144	0.999	5148	1.000
8142	1.000	8144	1.000	8148	1.000
3222	1.620	3224	2.178	3228	2.477
4222	1.030	4224	0.999	4228	1.000
5222	1.000	5224	1.000	5228	0.999
8222	1.000	8224	0.999	8228	0.999
3422	1.289	3424	1.000		
4422	1.133	4424	1.000		
5422	1.071	5424	0.999		
8422	0.961	8424	0.999		
3442	1.813				
4442	1.152				
5442	1.084				
8442	1.036				

Table 7.11: Factors for Square link model (organized by shape).

encryption algorithm works on a block of 64 bits and uses a key of 128 bits.

The authors' concept was to find the best non-linear transformation idea and the best diffusion transformation idea and then combine the two into a cipher. The idea was that combining two great ideas would result in a stronger product. However, this idea ignores how well the two ideas work together and the second and third order effects on the entire cipher (i.e., emergent behavior could compromise security).

Shark's non-linear transformation utilizes eight 8x8 bit non-linear S-boxes. These S-boxes also use the inverse function and then an invertible affine function to remove the fixed points in the inversion. Shark's diffusion transformation multiplies the state array by a MDS (Maximal Distance Separable) code. The purpose of this transformation is to provide an avalanche effect where a small change in the input to the round will cause a large change in the output. This is similar in function to the circulant matrix in the Mix Columns step of AES. A MDS code is chosen because they give an optimal branch number which translates into the greatest avalanche effect possible. Shark specifically uses a Reed-Soloman code in the transformation's implementation.

The key scheduling process for Shark appears to be quite complicated in their paper [39]. It is also quite expensive (time-wise) due to the overhead of adding extra entropy to the process.

nrce	factor	estimated time
3428	1.00	$268,261$ years
4428	1.00	$536,173$ years
5428	1.00	$982,690$ years
8428	1.00	$1,965,030$ years
3444	1.00	$273,867$ years
4444	1.00	$988,296$ years
5444	1.00	$1,702,724$ years
8444	1.00	$3,846,010$ years
3448	1.00	$4.949 \cdot 10^{24}$ years
4448	1.00	$1.813 \cdot 10^{25}$ years
5448	1.00	$3.131 \cdot 10^{25}$ years
8448	1.00	$7.084 \cdot 10^{25}$ years

Table 7.12: Estimated solution times for large Square variants.

However, it still looks like the keys are generated through a linear process like Square and not like AES. This should result in a weaker cipher.

7.3 Anubis

Anubis, designed by Vincent Rijmen and Paulo S. L. M. Barreto, is the child of AES. Anubis operates on blocks of 128 bits while its key length can range from 128 bits to 320 bits. However, the key length must increase in steps of 32 bits. Like AES, an increase in key size results in an increase in the number of rounds used in the encryption process. There are twelve rounds for a 128-bit key, thirteen rounds for a 160-bit key, and so forth resulting in eighteen rounds for a 320-bit key.

In Anubis, there are three transformations in each round. These are a non-linear transformation (using an S-box), a linear transformation, and a key addition. The non-linear transformation utilizes an S-box that is completely different from all previously discussed ciphers in this dissertation. This transformation pushes the state array through sixteen 8-bit by 8-bit S-boxes which can be performed in parallel. Smaller pseudo-random 4x4 boxes generate these mini S-boxes.. This shift in S-box usage within a cipher is done in order to make its hardware implementation easier while making its polynomial representation more complex. Anubis's linear transformation consists of a matrix transposition followed by multiplication by an MDS matrix. This step provides the cipher with diffusion and confusion. The key addition transposition is identical to the other ciphers in this work.

The key schedule algorithm for Anubis is different from AES and its predecessors. It utilizes a variant of the round function plus a projection over the key space. Other differences between Anubis and AES are that Anubis 1) uses a different polynomial to define the finite field, 2) uses entries from the S-box to define the round constant, and 3) only has one possible block size. Additionally, all transformations within the Anubis rounds are involutions. ***Involutions*** are bijective functions that are their own inverses. For a cipher, this means that the same software and hardware can be used to encrypt and decrypt. This should make the cipher implementation compact [40].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 8:
Conclusion

This dissertation sets out to explore the NP-Hard problem of solving systems of multi-variate polynomial equations that exist in a finite field. It analyzes the method of MRHS introduced by Raddum and Semaev in 2006 and expands their techniques to solve the system more efficiently. Our new techniques and knowledge are then applied to common encryption algorithms to showcase their capabilities. While we did not render modern encryption standards null and void, we did improve on existing work in the field and provide more evidence that AES's structure can play a significant role in its defeat.

In this final chapter, we review the dissertation's research contributions, as well as discuss directions for future research.

8.1 Contributions

The following are the dissertation's main research contributions. The first three highlight how we expand the innovative MRHS approach of Raddum and Semaev.

First, we modified the concept of MRHS equations to represent operations within the byte field instead of the bit field. This improvement significantly reduces the number of variables and equations within the target systems. It also clarifies the underlying structure of the system so that it can be used against itself. Representing these equations in terms of larger fields is a natural technique for ciphers such as AES. Specifically for full AES, this consisted of only 200 equations.

Second, we extended the concept of linking two equations to creating multi-links of any number of equations. While our method of multi-linking uses the same type of information as normal linking, we allow links of any number of equations, and thus gain information unavailable to normal pair-wise linking. This enhanced information leads to greater efficiency in solving the system.

Third, we extended the concept of agreeing one link at a time to multi-agreeing a growing set of multi-links. Once right-hand sides are eliminated via multi-agreeing, the system is quickly solved. This concept completely precludes the need for memory-limited gluing and guessing

operations. The trade-off is that our new approach is time-intensive due to the number of free equations in the first multi-agree that eliminates RHSs.

Fourth, we performed numerous computational experiments to explore how to apply these methods most effectively. This leads to the discovery of suitable multi-agreeing strategies for various shapes of AES variants, including a strategy for full AES. Once strategies are created for three or four rounds of a cipher, these strategies can easily extend to ten rounds without increasing the number of free equations. Surprisingly, we find many cases of AES variants that result in multiple solutions when the system is solved. The presence of multiple solutions means that multiple keys transform a plaintext to a single ciphertext.

Fifth, we executed the concept of multi-agreeing with multi-links on the cipher Square. To accomplish this, we develop strategies to solve both the small variants and full Square.

Sixth, we found a multi-agreeing strategy for full AES that only used sixteen free equations. This is comparable to using a brute force technique (trying all combinations of sixteen bytes for the key). We also develop a time model for how long this strategy would take to execute on a single computer processor. As far as we can determine, no other purely algebraic attack on AES has been shown to be this efficient.

8.2 Future Research

This body of work brings forth several lines of future research. These lines exist in the mathematical arena, the computing arena, and in applications.

There are five lines of future work which lie in the mathematical arena. First, the concept of optimizing the basis of the link space through the creation of custom-designed links should continue to be explored. Perhaps there exists a different set of multi-links for an AES shape that will solve the system more efficiently than by only using the default set of multi-links. An optimized basis might consist of multi-links with a reduced average number of equations in them in order to reduce the number of free equations in the multi-agreeing order. Second, additional exploration is needed in the search for optimal multi-agreeing orders. While the orders presented in this research are efficient, this does not preclude the existence of a more optimal order. Heuristics from the current computational experiments could be created to assist in this endeavor.

Third, our methods and strategies could be expanded to the case of related keys. Many modern

cryptanalysis methods are designed for use on special cases of encryption where additional data is known. The related key case, where a plaintext is encrypted by multiple keys that are related in some fashion, is frequently studied. This construct creates additional MRHS equations that could make the system faster to solve. Fourth, the concept of using multiple plaintext/ciphertext pairs of information could still be explored. Perhaps there are occasions on which this concept could assist in solving the system. Finally, this research noted that many cases of the small scale AES variants yielded multiple keys that encrypt a given plaintext to a single ciphertext. Exploration could be conducted to determine if there are relationships between certain keys or plaintext inputs that allow this to occur.

There are two lines of future work in the computing arena. First, the computationally intensive multi-agree that first eliminates RHSs could easily be parallelized over many processors. Second, the agreeing tree concept can be coded to allow a computer to determine if the agreeing orders created in this research can be tweaked in order to become more efficient.

Lastly, this method can also be applied to other problems in future work. Our techniques can be applied to other symmetric block ciphers to analyze their strength. Also, these techniques are not limited to cryptosystems. There are many other applications in other fields of study that can be described by a system of multi-variate polynomial equations over a finite field.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Pseudo-Code for Multi-agree

Pseudo-code of the multi-agree algorithm

When the multi-agree algorithm reaches a check-for-consistency link—we call this kablooie!:

```
kablooie!
  initialize combo of indices to RHS for each eq, and counters

checkcombo:  // loop back here to try next combo
  didcombos++;
  for ( each unagreed link ) {
    didlinks++;
    loop to add up z values, except last
    if ( indexed link ) {   // last eq in link is new, indexed by z value
      look up eq rhs index for this z
      if ( valid rhs idx ) save it to combo
      else if ( no rhs for that z ) goto checkskipped; //link does not agree
      else {have repeated z value; set up to track repetition;
               save rhs idx to combo}
    } else   // else this link is just a consistency check
      if ( z sum does not match z val of last eq )
          goto checkskipped; // link does not agree
  } // at this point, this link agrees; check next link

agreed:  // at this point, have agreed set: count it; flag needed rhs
  loop to flag each rhs in agreed set as needed
  goto nextcombo;   // jump over next section

checkskipped: //come here when link does not agree, reset any skipped eqs
  if ( skipped over some variable eqs ) reset combo for skipped eqs

nextcombo:  // find next combo of rhs indices (for variable eqs) to try
```

```
  do {  // seek next combo of rhs, starting at current eq in current link
    if ( eq indexed ) {  // then may be repeated zs
      if ( repeats remain for same z ) {
        use next rhs for same z; this link OK, skip to next
        goto checkcombo;  // loop back to top of outer loop
      }  // at this point, no more repeats for this z, loop for prev eq
    } else  // variable eq
      if ( haven't tried all rhs for this eq ) {
        try next rhs;
        goto checkcombo;  // loop back to top of outer loop
      }
      else // no more rhs for this eq;
        reset this eq;  // and loop for prev eq
  } while ( got another combo to check );
//  at this point, have tried all combos
```

APPENDIX B:
Agreeing Order for $A44e$

This is the multi-agreeing order used for the case of $A44e$. There are sixteen degrees of freedom so there are sixteen check-for-consistency links within this order.

Table B.1: Agreeing order for $A44e$.

link number	new eqns	# degrees of freedom	comment
Link #51	2,43	1	
Link #21	6,7,10,11,14,40,41,42,63	9	
Link #115	51	9	
Link #147	55	9	
Link #83	3,47	10	
Link #123	52	10	
Link #59	4,44	11	
Link #75	1,46	12	
Link #23	17	12	
Link #43	13	12	
Link #67	0,9,12,45	15	
Link #37	67	15	
Link #55	19	15	
Link #87	16	15	
Link #99	49	15	
Link #119	20	15	
Link #139	54	15	
Link #163	57	15	
Link #15	27	15	
Link #39	22	15	
Link #63	30	15	
Link #71	23	15	
Link #111	33	15	

Continued on next page

Table B.1: Agreeing order for $A44e$.

link number	new eqns	# degrees of freedom	comment
Link #135	25	15	
Link #159	36	15	
Link #167	26	15	
Link #31	28	15	
Link #47	29	15	
Link #95	32	15	
Link #143	35	15	
Link #11	38	15	
Link #19	39	15	
Link #107	5,50	16	
Link #35	8	16	
Link #33	66	16	
Link #27	15	16	
Link #91	48	16	
Link #131	53	16	
Link #1	58	16	
Link #5	59	16	
Link #155	56	16	
Link #9	60	16	
Link #13	61	16	
Link #17	62	16	
Link #25	64	16	
Link #29	65	16	
Link #41	68	16	
Link #45	69	16	
Link #49	70	16	
Link #53	71	16	
Link #57	72	16	
Link #61	73	16	
Link #65	74	16	
Link #69	75	16	

Continued on next page

Table B.1: Agreeing order for $A44e$.

link number	new eqns	# degrees of freedom	comment
Link #73	76	16	
Link #77	77	16	
Link #81	78	16	
Link #85	79	16	
Link #89	80	16	
Link #93	81	16	
Link #97	82	16	
Link #101	83	16	
Link #105	84	16	
Link #109	85	16	
Link #113	86	16	
Link #117	87	16	
Link #121	88	16	
Link #125	89	16	
Link #129	90	16	
Link #133	91	16	
Link #137	92	16	
Link #141	93	16	
Link #145	94	16	
Link #149	95	16	
Link #153	96	16	
Link #157	97	16	
Link #161	98	16	
Link #165	99	16	
Link #0	100	16	
Link #2	101	16	
Link #4	102	16	
Link #6	103	16	
Link #8	104	16	
Link #10	105	16	
Link #12	106	16	

Continued on next page

Table B.1: Agreeing order for $A44e$.

link number	new eqns	# degrees of freedom	comment
Link #14	107	16	
Link #16	108	16	
Link #18	109	16	
Link #20	110	16	
Link #22	111	16	
Link #24	112	16	
Link #26	113	16	
Link #28	114	16	
Link #30	115	16	
Link #32	116	16	
Link #34	117	16	
Link #36	118	16	
Link #38	119	16	
Link #151	18	16	
Link #40	120	16	
Link #42	121	16	
Link #44	122	16	
Link #46	123	16	
Link #48	124	16	
Link #50	125	16	
Link #52	126	16	
Link #54	127	16	
Link #56	128	16	
Link #58	129	16	
Link #60	130	16	
Link #62	131	16	
Link #64	132	16	
Link #66	133	16	
Link #68	134	16	
Link #70	135	16	
Link #7	21	16	

Continued on next page

Table B.1: Agreeing order for $A44e$.

link number	new eqns	# degrees of freedom	comment
Link #72	136	16	
Link #74	137	16	
Link #76	138	16	
Link #78	139	16	
Link #80	140	16	
Link #82	141	16	
Link #84	142	16	
Link #86	143	16	
Link #88	144	16	
Link #90	145	16	
Link #92	146	16	
Link #94	147	16	
Link #96	148	16	
Link #98	149	16	
Link #100	150	16	
Link #102	151	16	
Link #103	24	16	
Link #104	152	16	
Link #106	153	16	
Link #108	154	16	
Link #110	155	16	
Link #112	156	16	
Link #114	157	16	
Link #116	158	16	
Link #118	159	16	
Link #120	160	16	
Link #122	161	16	
Link #124	162	16	
Link #126	163	16	
Link #128	164	16	
Link #130	165	16	

Continued on next page

Table B.1: Agreeing order for $A44e$.

link number	new eqns	# degrees of freedom	comment
Link #132	166	16	
Link #134	167	16	
Link #79	31	16	
Link #136	168	16	
Link #138	169	16	
Link #140	170	16	
Link #142	171	16	
Link #168	184	16	
Link #169	–	15	check-for-consistency
Link #182	191	15	
Link #183	–	14	check-for-consistency
Link #127	34	14	
Link #3	37	14	
Link #188	194	14	
Link #189	–	13	check-for-consistency
Link #194	197	13	
Link #195	–	12	check-for-consistency
Link #144	172	12	
Link #146	173	12	
Link #148	174	12	
Link #150	175	12	
Link #170	185	12	
Link #171	–	11	check-for-consistency
Link #176	188	11	
Link #177	–	10	check-for-consistency
Link #190	195	10	
Link #191	–	9	check-for-consistency
Link #196	198	9	
Link #197	–	8	check-for-consistency
Link #152	176	8	
Link #154	177	8	

Continued on next page

Table B.1: Agreeing order for $A44e$.

link number	new eqns	# degrees of freedom	comment
Link #156	178	8	
Link #158	179	8	
Link #172	186	8	
Link #173	–	7	check-for-consistency
Link #178	189	7	
Link #179	–	6	check-for-consistency
Link #184	192	6	
Link #185	–	5	check-for-consistency
Link #198	199	5	
Link #199	–	4	check-for-consistency
Link #160	180	4	
Link #162	181	4	
Link #164	182	4	
Link #166	183	4	
Link #174	187	4	
Link #175	–	3	check-for-consistency
Link #180	190	3	
Link #181	–	2	check-for-consistency
Link #186	193	2	
Link #187	–	1	check-for-consistency
Link #192	196	1	
Link #193	–	0	solved with this check

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C:
AES MRHS Equation Creation Code

This appendix contains the C code for the construction of AES MRHS equations. It consists of a main file (aes_eqs_encr_f_blks) and a header file (eqs_io_f_blks).

aes_eqs_encr_f_blks.c

```
/*
        aes_eqs.c
        version: 2012 Apr 20
Generate MRHS Equations for
Small Scale Variants of the AES algorithm
Also does the encryption!
Output in field elements, not bits!
Handles multiple blocks!!
Notes: always uses * form: last round no MixCols
        always keysize = block size
optional command line arguments:
        variant (string) = "nrce" to specify small-scale variant of AES:
                        n (hex) is # rounds ( 1 - A; default=A=10 )
                        r (int) is # rows ( 1, 2, 4; default=4 )
                        c (int) is # cols ( 1, 2, 4; default=4 )
                        e (int) is # bits in word ( 2, 4, 8; default=8 )
                        defaults are "A448" for standard AES = SR*(10; 4; 4; 8)
        nblocks (int) = number of plaintext blocks to encrypt (default = 1)
        key (hex) = key block (default is zero block)
        input (string) = filename of input file of plaintext (default is NULL)
while all the above are optional, you must have one to have the next...
If no input file is specified, random blocks will be generated.
output goes to stdout

save all the X state data (output of S-box after ShiftRows) and K key data,
```

```
print it out after the equations.
encryption re−organized the to give the X state:
        put ShiftRows before S−box, as part of previous round
        do NOT do "in place"; rather, put result in new place.
(ARS) (MARS)∗(n−1) (A) rather than
(A) (SRMA)∗(n−1) (SRA) [where R is RowShift, S is SubstBytes, ...]
Does actual KeySchedule and Encrypt.
(Note: keep InvMix in case do non−star versions.)
∗/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//#define OUTPUTBITS // to print output in bits (otherwise in field elements)

#define MAXROUNDS 10
#define MAXROWS 4
#define MAXCOLS 4
#define MAXBITS 8
#define MAXNBLOCKS 16 // number of plaintext blocks
#define MAXBLOCK MAXROWS∗MAXCOLS
#define MAXKEY MAXBLOCK
#define MAXVARS MAXBLOCK+MAXROUNDS∗MAXROWS + MAXNBLOCKS∗ (
    MAXROUNDS−1)∗MAXBLOCK
#define SR(c,r) ( (c+r) & (nCols−1) )

FILE ∗infile;
unsigned char RoundKeys[(MAXROUNDS + 1) ∗ MAXBLOCK];
unsigned char BlockStates[MAXNBLOCKS∗ (MAXROUNDS + 2) ∗ MAXBLOCK];
unsigned int ∗Log;
unsigned char ∗ALog, ∗Sbox, ∗Mix, ∗InvMix, fieldmask;
int nRounds = 10, nRows = 4, nCols = 4, nBits = 8, star = 1, field, block,
    nBlocks = 1,
    KeyBits, KeyCols, nKeyCols;
```

```
int nEqs, nVars, nbVars, nKeyVars, bstates, CurrentVars;
unsigned char blkbuf[MAXBLOCK], *CT, *PT, Eq[2][MAXVARS], Data[2];
enum InOut { In, Out };
enum VarType { Key, X, State };

unsigned int Log8[256] = {
0x00,0x00,0x19,0x01,0x32,0x02,0x1A,0xC6,0x4B,0xC7,0x1B,0x68,0x33,0xEE,0xDF,0x03,
0x64,0x04,0xE0,0x0E,0x34,0x8D,0x81,0xEF,0x4C,0x71,0x08,0xC8,0xF8,0x69,0x1C,0xC1,
0x7D,0xC2,0x1D,0xB5,0xF9,0xB9,0x27,0x6A,0x4D,0xE4,0xA6,0x72,0x9A,0xC9,0x09,0x78
    ,
0x65,0x2F,0x8A,0x05,0x21,0x0F,0xE1,0x24,0x12,0xF0,0x82,0x45,0x35,0x93,0xDA,0x8E,
0x96,0x8F,0xDB,0xBD,0x36,0xD0,0xCE,0x94,0x13,0x5C,0xD2,0xF1,0x40,0x46,0x83,0x38,
0x66,0xDD,0xFD,0x30,0xBF,0x06,0x8B,0x62,0xB3,0x25,0xE2,0x98,0x22,0x88,0x91,0x10,
0x7E,0x6E,0x48,0xC3,0xA3,0xB6,0x1E,0x42,0x3A,0x6B,0x28,0x54,0xFA,0x85,0x3D,0xBA
    ,
0x2B,0x79,0x0A,0x15,0x9B,0x9F,0x5E,0xCA,0x4E,0xD4,0xAC,0xE5,0xF3,0x73,0xA7,0x57
    ,
0xAF,0x58,0xA8,0x50,0xF4,0xEA,0xD6,0x74,0x4F,0xAE,0xE9,0xD5,0xE7,0xE6,0xAD,0
    xE8,
0x2C,0xD7,0x75,0x7A,0xEB,0x16,0x0B,0xF5,0x59,0xCB,0x5F,0xB0,0x9C,0xA9,0x51,0
    xA0,
0x7F,0x0C,0xF6,0x6F,0x17,0xC4,0x49,0xEC,0xD8,0x43,0x1F,0x2D,0xA4,0x76,0x7B,0xB7,
0xCC,0xBB,0x3E,0x5A,0xFB,0x60,0xB1,0x86,0x3B,0x52,0xA1,0x6C,0xAA,0x55,0x29,0
    x9D,
0x97,0xB2,0x87,0x90,0x61,0xBE,0xDC,0xFC,0xBC,0x95,0xCF,0xCD,0x37,0x3F,0x5B,0
    xD1,
0x53,0x39,0x84,0x3C,0x41,0xA2,0x6D,0x47,0x14,0x2A,0x9E,0x5D,0x56,0xF2,0xD3,0xAB,
0x44,0x11,0x92,0xD9,0x23,0x20,0x2E,0x89,0xB4,0x7C,0xB8,0x26,0x77,0x99,0xE3,0xA5,
0x67,0x4A,0xED,0xDE,0xC5,0x31,0xFE,0x18,0x0D,0x63,0x8C,0x80,0xC0,0xF7,0x70,0x07,
};

unsigned char ALog8[256] = {
0x01,0x03,0x05,0x0F,0x11,0x33,0x55,0xFF,0x1A,0x2E,0x72,0x96,0xA1,0xF8,0x13,0x35,
0x5F,0xE1,0x38,0x48,0xD8,0x73,0x95,0xA4,0xF7,0x02,0x06,0x0A,0x1E,0x22,0x66,0xAA,
```

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

unsigned int Log4[16] = {
0, 0, 1, 4, 2, 8, 5, 10, 3, 14, 9, 7, 6, 13, 11, 12,
};

unsigned char ALog4[16] = {
1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9, 1,
};

unsigned int Log2[4] = {
0, 0, 1, 2,
};
```

```
unsigned char ALog2[4] = {
1, 2, 3, 1,
};

unsigned char Sbox8[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,0xA4,0x72,0
    xC0,
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,0x58,0
    xCF,
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,0
    x08,
0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0
    x8A,
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,
};

unsigned char Sbox4[16] = {
0x6,0xB,0x5,0x4,0x2,0xE,0x7,0xA,0x9,0xD,0xF,0xC,0x3,0x1,0x0,0x8,
};

unsigned char Sbox2[4] = {
 2, 3, 1, 0,
```

```
};

unsigned char Mix4[4] = {
0x2,0x3,0x1,0x1,
};

unsigned char InvMix4[4] = {
0xE,0xB,0xD,0x9,
};

unsigned char InvMix42[4] = {
0x0,0x2,0x3,0x0,
};

unsigned char Mix2[2] = {
0x3,0x2,
};

unsigned char Mix1[1] = {
0x1,
};

// multiply by "2" in field
#define POLY8 0x1B
#define POLY4 0x13
#define POLY2 0x07
unsigned char HIBIT, POLY;
unsigned char mul2 (unsigned char x) {
    unsigned char y;
    y = x << 1;
    if ( x & HIBIT ) y ^= POLY;
    return( y );
}
```

```
// multiply two bytes in field
unsigned char mul(unsigned char x, unsigned char y)
{
    if (x && y)
        return (ALog[(Log[x] + Log[y]) % (field − 1)]);
    else
        return (0);
}

#include "eqs_io_f_blks.h" // include field I/O package

// set up specific small−scale variant of AES
// assumes main() already set: nRounds, nRows, nCols, nBits, star
int setup(void)
{
    int returnval = 0;

// check parameters for validity
    if (nBlocks < 1 || nBlocks > MAXNBLOCKS) {
        nBlocks = 1;
        returnval = 1;
    }
    if (nRounds < 1 || nRounds > MAXROUNDS) {
        nRounds = 10;
        returnval = 1;
    }
    if (!(nCols == 1 || nCols == 2 || nCols == 4)) {
        nCols = 4;
        returnval = 1;
    }

    switch (nBits) {
    case 2:
        Log = Log2;
```

```
        ALog = ALog2;
        Sbox = Sbox2;
        POLY = POLY2;
        field = 4;
        break;
    case 4:
        Log = Log4;
        ALog = ALog4;
        Sbox = Sbox4;
        POLY = POLY4;
        field = 16;
        break;
    default:
        nBits = 8;
        returnval = 1;  // if bad value, use default, fall thru
    case 8:
        Log = Log8;
        ALog = ALog8;
        Sbox = Sbox8;
        POLY = POLY8;
        field = 256;
        break;
    }

    switch (nRows) {
    case 1:
        Mix = InvMix = Mix1;
        break;
    case 2:
        Mix = InvMix = Mix2;
        break;
    default:
        nRows = 4;
        returnval = 1;  // if bad value, use default, fall thru
```

```
        case 4:
            Mix = Mix4;
            InvMix = (nBits == 2) ? InvMix42 : InvMix4;
            break;
        }
        fieldmask = field − 1;
        HIBIT = 1 << (nBits − 1);
        setScale(); // set up bit matrices for scalars
        block = nRows ∗ nCols;
        bstates = (nRounds + 2) ∗ block;
        KeyBits = block ∗ nBits;
        nKeyCols = (nRounds + 1) ∗ nCols;
        nKeyVars = block + nRounds ∗ nRows;
        nbVars = block ∗ (nRounds − 1);
        nVars = nKeyVars + nBlocks ∗ nbVars;
        nEqs = nVars + (nBlocks−1) ∗ block;
    srand( (unsigned)( nRows | nCols<<2 | nBits<<4 | nRounds<<8 | nBlocks << 12) );

        return returnval;
}

int KeySchedule(unsigned char Key[])
{
        int colbits, returnval = 0;
        int r, c;
        unsigned char col[MAXROWS], t, rcon;

        colbits = nRows ∗ nBits;
        KeyCols = KeyBits / colbits;
#define NOISY 0
#if NOISY
fprintf(stderr,"−KeySched:␣colbits=%d,␣KeyCols=%d,␣nKeyCols=%d,␣Key=%p\n",
  colbits, KeyCols, nKeyCols, Key);
fprintf(stderr,"−Key:␣");
```

```
 for (r = 0; r < block; r++) fprintf(stderr, ((nBits>4) ? "%02X" : "%01X"), Key[r]);
fprintf(stderr,"\n"); fflush(stderr);
#endif
    /* Copy key */
    for (c = 0; c < KeyCols; c++)
        for (r = 0; r < nRows; r++)
            RoundKeys[r + nRows * c] = Key[r + nRows * c];
    for (r = 0; r < nRows; r++)
        col[r] = Key[r + nRows * (c - 1)];
#if NOISY
fprintf(stderr,"-KeySched: Key copied; c=%d, col= ", c);
 for (r = 0; r < nRows; r++) fprintf(stderr, (nBits>4)?"%02X":"%01X", col[r]);
fprintf(stderr,"\n"); fflush(stderr);
#endif

    for (rcon = 1; c < nKeyCols; c++) {
        /* calculate new columns until enough */
        if (c % KeyCols == 0) {
            t = col[0];
            for (r = 0; r < (nRows - 1); r++)
                col[r] = Sbox[col[r + 1]];
            col[nRows - 1] = Sbox[t];
            col[0] ^= rcon;
#if NOISY
fprintf(stderr,"-KeySched: apply F; t=%X, rcon=%X, col= ", t, rcon);
 for (r = 0; r < nRows; r++) fprintf(stderr, (nBits>4)?"%02X":"%01X", col[r]);
fprintf(stderr,"\n"); fflush(stderr);
#endif
            rcon = mul(2, rcon);
        }
// need to handle KeyCols = 1 differently
        for (r = 0; r < nRows; r++)
            RoundKeys[r + nRows * c] = (KeyCols == 1) ? col[r] :
                (col[r] ^= RoundKeys[r + nRows * (c - KeyCols)]);
```

```
#if 0
fprintf(stderr,"−KeySched:␣RoundKeys␣col[%d]=␣", c);
 for (r = 0; r < nRows; r++) fprintf(stderr, (nBits>4)?"%02X":"%01X",
 RoundKeys[r + nRows ∗ c]);
fprintf(stderr,"\n"); fflush(stderr);
#endif
    }

    return returnval;
}

// do one round on block: (ARS) for #0 or else (MARS)
void doround(unsigned char State[], unsigned char roundKey[],
             int round)
{
    unsigned char t[MAXROWS];
    int i, r, c, offset=0;
    if (round) // if normal round
        for (c = 0; c < nCols; c++) {
            for (r = 0; r < nRows; r++)
                for (t[r] = i = 0; i < nRows; i++)
                    t[r] ^= // MixColumns
                        mul(Mix[i], State[((r + i) % nRows) + nRows ∗ c]);
            for (r = 0; r < nRows; r++)
                State[block + r + nRows ∗ c] = t[r];
        }
else offset = −block;
State += block;
    for (i = 0; i < block; i++)
        State[i] = State[offset+i] ^ roundKey[i]; // AddRoundKey
    for (r = 1; r < nRows; r++) {
        for (c = 0; c < nCols; c++) // ShiftRows
            t[c] = State[r + nRows ∗ ((c + r) % nCols)];
        for (c = 0; c < nCols; c++)
```

```
                State[r + nRows * c] = t[c];
        }
    for (i = 0; i < block; i++)
            State[i] = Sbox[State[i]]; // SubBytes

}

// do round #n on block: (A)
void doroundn(unsigned char State[], unsigned char roundKey[])
{
    int i;

    for (i = 0; i < block; i++)
            State[block + i] = State[i] ^ roundKey[i]; // AddRoundKey
}

// encrypt block (NOT in place - keep output of each S-box)
void encrypt( unsigned char States[] )
{
    int i, round;

    for (i = 0; i < block; i++)
            States[i] = PT[i];  // copy PT in
    for (round = 0; round < nRounds; round++) {
            doround( States + round*block, RoundKeys + round*block, round);
    }
    doroundn(States + round*block, RoundKeys + round*block);
}

void NewEq( void )
{
    int i, r;

    for (r = In; r <= Out; r++) {
```

```
    Data[r] = 0;
    for (i = 0; i < nVars; i++)
        Eq[r][i] = 0;
    }
}

int VarNum( enum VarType var,
        int round, int col, int row )
{
    switch (var) {
    case Key:
        if (round == 0)
            return ( col * nRows + row );
        else // then col == 0
            return ( block + (round-1) * nRows + row );
    case X:
        return ( CurrentVars + (round-1) * block + col * nRows + row );
    }
    return ( 0 ); // dummy
}

void AddVar( enum InOut line, enum VarType var,
        int round, int col, int row, int scale )
{
    int r;

    switch (var) {
    case Key:
        if (round == 0 || col == 0)
            Eq[line][ VarNum( Key, round, col, row ) ] ^= scale;
        else {
            AddVar( line, Key, round, col-1, row, scale );
            AddVar( line, Key, round-1, col, row, scale );
        }
```

```
            break;
        case X:
            Eq[line][ VarNum( X, round, col, row ) ] ^= scale;
            break;
        case State: // scale must be 1
            for (r = 0; r < nRows; r++) {
                AddVar( line, X, round, col, (row+r)&(nRows−1), Mix[r] );
            }
            AddVar( line, Key, round, col, row, 1 );
            break;
        }
}

void KeyScheduleEqs(void)
{
    int i, r;
    unsigned char rcon;

    for (i = 1, rcon = 1; i <= nRounds; i++) {
        for (r = 0; r < nRows; r++) {
            NewEq();
            AddVar( Out, Key, i, 0, r, 1);
            if ( nCols > 1 ) AddVar( Out, Key, i−1, 0, r, 1);
            AddVar( In, Key, i−1, nCols−1, (r+1)&(nRows−1), 1);
            if ( r == 0 ) Data[ Out ] = rcon;
            WriteEq();
        }
        rcon = mul2(rcon);
    }
}

// do only 1 round on block
void doonlyroundEqs(int round)
{
```

```
    int r, c;

    for (c = 0; c < nCols; c++) {
        for (r = 0; r < nRows; r++) {
            NewEq();
            AddVar( Out, Key, round, c, r, 1);
            Data[ Out ] = CT[ r + nRows * c ];
            AddVar( In, Key, round-1, SR(c,r), r, 1);
            Data[ In ] = PT[ r + nRows * SR(c,r) ];
            WriteEq();
        }
    }
}

// do round #1 on block
void doround1Eqs(int round)
{
    int r, c;

    for (c = 0; c < nCols; c++) {
        for (r = 0; r < nRows; r++) {
            NewEq();
            AddVar( Out, X, round, c, r, 1);
            AddVar( In, Key, round-1, SR(c,r), r, 1);
            Data[ In ] = PT[ r + nRows * SR(c,r) ];
            WriteEq();
        }
    }
}

// do one round on block
void doroundEqs(int round)
{
    int r, c;
```

```
    for (c = 0; c < nCols; c++) {
        for (r = 0; r < nRows; r++) {
            NewEq();
            AddVar( Out, X, round, c, r, 1);
            AddVar( In, State, round−1, SR(c,r), r, 1);
            WriteEq();
        }
    }
}

// do round #n on block
void doroundnEqs(int round)
{
    int r, c;

    for (c = 0; c < nCols; c++) {
        for (r = 0; r < nRows; r++) {
            NewEq();
            AddVar( Out, Key, round, c, r, 1);
            AddVar( In, State, round−1, SR(c,r), r, 1);
            Data[ Out ] = CT[ r + nRows ∗ c ];
            WriteEq();
        }
    }
}

void EncryptEqs(void)
{
    int round;

    if ( nRounds == 1 ) {
        doonlyroundEqs(1);
        return;
```

```
        }
    doround1Eqs(1);
    for (round = 2; round < nRounds; round++) {
        doroundEqs(round);
    }
    doroundnEqs(round);
}

int main(int argc, char *argv[])
{
unsigned char Key[MAXBLOCK];
int b;

    if (argc > 1) {
        sscanf(argv[1], "%1x%1d%1d%1d",
                    &nRounds, &nRows, &nCols, &nBits);
    }
    if (argc > 2) {
        sscanf(argv[2], "%d",
                    &nBlocks);
    }
    fprintf(stderr, "␣nBlocks=%d,␣nRounds=%d,␣nRows=%d,␣nCols=%d,␣nBits=%d,␣star
        =%d\n",
            nBlocks, nRounds, nRows, nCols, nBits, star);
    if (setup())
        fprintf(stderr,
                "Bad␣parameter(s);␣now:\n␣nBlocks=%d,␣nRounds=%d,␣nRows=%d,␣
                    nCols=%d,␣nBits=%d,␣star=%d\n",
                nBlocks, nRounds, nRows, nCols, nBits, star);
// by default KeyBits = bits in block
        ReadBlock( (argc > 3) ? argv[3] : "", Key);
    if (argc > 4) {
        if ((infile = fopen(argv[4], "rb")) == NULL) {
            fprintf(stderr, "Could␣not␣open␣input␣file␣%s\n", argv[4]);
```

```
        }} else infile = NULL;

// do encryption
   PT = blkbuf;
   KeySchedule(Key);
      for ( b=0; b< nBlocks; b++ ) {
          NextBlock(); // puts next PT block into "PT"
          encrypt( BlockStates+(b*bstates) );
          }

// make system of eqs
      WriteSystemHeader();
      KeyScheduleEqs();
// do only single block
      for ( b=0; b < nBlocks; b++ ) {
          CurrentVars = nKeyVars + b*nbVars; // global var for indexing vars
          PT = BlockStates+(b*bstates); // point to PT block
          CT = BlockStates+((b+1)*bstates - block); // point to CT block
          EncryptEqs();
          }

      WriteVars();
      printf("␣nRounds=%d,␣nRows=%d,␣nCols=%d,␣nBits=%d,␣star=%d\n",
              nRounds, nRows, nCols, nBits, star);
      WriteKeys();
      WriteStates();

      return (0);
}
```

eqs_io_f_blks.h

```
/*
        eqs_io.h
        version: 2012 Apr 15
```

```
*/

void NextBlock(void) // generate random data blocks
{
    int i;
    for ( i=0; i< block; i++ ) PT[i] = rand() & fieldmask;
}



/*
Write Equation, field elements version
*/

char *FieldFormat;
void setScale(void){ // set up FieldFormat
  FieldFormat = (nBits > 4)? "%02X" : "%01X";
}

void writeValue(unsigned int x){
  /* writes field element x */

      printf( FieldFormat, x );
}

void WriteSystemHeader( void ) // note third number on top line: #bits in field
{
    printf("  %d  %d  %d\n", nVars, nEqs, nBits );
}

void WriteEq( void )
{
    int i, r;

    printf(" %d %d\n", 2, field);
```

```
    for (r = In; r <= Out; r++) {
    for (i = 0; i < nVars; i++)
        writeValue( Eq[r][i] );
    printf("\n");
    }
    for ( i=0; i < field; i++ ) {
        if ( !(i&15) && i ) printf("\n");
        writeValue( i ^ Data[In] );
        writeValue( Sbox[i] ^ Data[Out] );
        }
        printf("\n");
}

// read hex data into block
void ReadBlock( char str[], unsigned char T[] )
{
    char *j;
    int i, word;

    switch (nBits) {
    case 2:
    if ( block == 1 ) { // this is the only case with an odd number of words
        word = 0;
        sscanf( str, "%1x", &word );
        T[0] = (unsigned char) (word & fieldmask);
        i = 1; break;
        }
        for (i = 0, j = str; i < block; ) {
            if ( sscanf( j++, "%1x", &word ) != 1 ) break;
            T[i++] = (unsigned char) ((word>>2) & fieldmask);
            T[i++] = (unsigned char) (word & fieldmask);
        }
        break;
    case 4:
```

```
            for (i = 0; i < block; i++) {
                if ( sscanf( str+i, "%1x", &word ) != 1 ) break;
                T[i] = (unsigned char) word;
            }
            break;
        case 8:
            for (i = 0; i < block; i++) {
                if ( sscanf( str+2*i, "%2x", &word ) != 1 ) break;
                T[i] = (unsigned char) word;
            }
            break;
        default:
                    i = 0;
        }
            for ( ; i < block; i++)
                T[i] = 0;
}

// write hex data from block
void WriteBlock( unsigned char T[] )
{
        int i;

            for (i = 0; i < block; i++)
                writeValue( T[i] & fieldmask );
}

// write Round Keys
void WriteKeys( void )
{
        int round;

        printf( "Round␣Keys:\n");
        for (round = 0; round <= nRounds; round++) {
```

```
                printf( "␣%2d␣:␣", round);
                WriteBlock( RoundKeys + round∗block );
                printf("\n");
                }
}

// write States
void WriteStates( void )
{
    int b, round;

    printf( "States:\n");
    for (b = 0; b < nBlocks; b++) {
                printf( "␣Block␣%2d␣:␣\n", b);
    for (round = 0; round <= nRounds+1; round++) {
                printf( "␣%2d␣:␣", round);
                WriteBlock( BlockStates+b∗bstates + round∗block );
                printf("\n");
                }}
}

// write Variables
void WriteVars( void )
{
    int i, b, round;

    printf( "Variables:\n");
        for (i = 0; i < block; i++)
                writeValue( RoundKeys[i] );
                printf("\n");
    for (round = 1; round <= nRounds; round++) {
        for (i = 0; i < nRows; i++)
                writeValue( RoundKeys[round∗block + i] );
                printf("\n");
```

```
                }
    for (b = 0; b < nBlocks; b++) {
    for (round = 1; round < nRounds; round++) {
        for (i = 0; i < block; i++)
            writeValue( BlockStates[b∗bstates + round∗block + i] );
        printf("\n");
        }}
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D:
AES MRHS Algorithm Code

This appendix contains the C code for the manipulation of AES MRHS equations as detailed in this body of work. It consists of a main file (main) and three header files (fieldarith, newagree, newmrhs).

main.c

```
/*
  new mrhs main.c
  version: 2013 Feb 23
  Interactive tool to solve MRHS equations

  input file format:
  header:
  Nvar # variables
  neqs # equations (symbols)
  nbits # field size in bits
  for each equation symbol:
  nrows # rows
  nrhs # RHS
  A (by rows, field elements)
  B (by cols, field elements)

  */

#include "newmrhs.h"    /* includes all others */
#include <ctype.h>

#define SPECIAL -30000 // a flag value

int getparam(char input[], int special) {
```

```
  int i;

  if (input[1])
    i = 1;
  else {
    scanf("%100s", input);
    i = 0;
  }
  if (sscanf(input + i, "%d", &i) != 1)
    i = special;
  return i ;
}

int main (int argc, const char * argv[]) {
  int i, j, k, n, *eqlist;
  Status rv;
  char input[101], c, *filename;
  const char menu[] =
  "Enter␣commands␣from␣the␣menu␣below.\n"
  "␣␣#␣means␣a␣number␣is␣required:␣use␣nondigit␣for␣default␣value␣(or␣for␣'all')\n"
  "r␣#␣ReduceEqs␣(number,␣then␣indices)\n"
  "l␣#␣LinkEqs␣(number␣[default:␣all],␣then␣indices)\n"
  "n␣#␣newagree␣(Link␣#␣[default:␣0])\n"
  "x␣␣␣print␣solutions␣from␣new-agreed␣sets\n"
  "z␣#␣print␣z␣value␣matrix␣of␣link␣(Link␣#␣[def.␣0])\n"
  "i␣#␣EquationInfo␣E[#]␣[default:␣all]\n"
  "p␣#␣print␣Equation␣E[#]\n"
  "e␣␣␣print␣LinEquation␣LE\n"
  "k␣#␣print␣Link␣E[#]␣[default:␣all]\n"
  "d␣#␣disagree␣(Link␣#␣[def.␣0],␣then␣RHS␣indices)\n"
  "f␣#␣find␣RHS␣that␣agree␣(Link␣#␣[def.␣0],␣then␣Eq␣#,␣RHS␣indices␣for␣others)\n
      "
  "w␣␣␣WriteSystem(S);\n"
  "v␣␣␣WriteLinkVars(S);\n"
```

```
  "y␣␣␣WriteLinkEqs(S);\n"
  "g␣␣␣WriteAgreedLinks(S);\n"
  "q␣␣␣QUIT␣(exit)\n"
  "*␣␣␣multiply␣field␣elements␣(hex)\n"
  "−␣␣␣invert␣field␣element␣(hex)\n"
  "/␣␣␣divide␣field␣elements␣(hex)\n"
  "^␣␣␣field␣element␣to␣power␣(hex,␣int)\n"
  "+␣␣␣add␣field␣elements␣(hex)\n"
  ;

#ifdef SQUARE
  fprintf(stderr, "␣using␣SQUARE␣arithmetic\n");
#else
  fprintf(stderr, "␣using␣AES␣arithmetic\n");
#endif

  S = (Sys *) malloc(sizeof(Sys));
  /*
   read in header, set up sys, setup & read eqns
   write system
   */

  if (argc > 1)
    filename = (char *) argv[1];
  else {
    fprintf(stderr, "Enter␣filename␣for␣input␣system:␣");
    scanf("%100s", input);
    filename = input;
  }
  if (ReadSystem(S, filename)) {
    fprintf(stderr, "Error␣in␣main:␣did␣not␣get␣input␣system;␣abort!\n");
    return (1);
  }
  new_init_agreed( );
```

```
printf("got:␣Nvar␣=␣%d;␣neqs␣=␣%d;␣Nbits␣=␣%d\n", Nvar, S−>neqs, Nbits);

eqlist = (int ∗) malloc(S−>neqs ∗ sizeof(int));

printf("%s",menu);
printf(">␣");
while (scanf("%100s", input) == 1) {
  rv = OK;
  switch (c = input[0]) {

    case 'a':
    case 'n':
      i = getparam( input, 0 );
      if (i < 0) { i = −i; k = 1; }
      else if (input[1]=='−') k = 1;
      else k = 0;
      if (i >= S−>nlinks) {
        fprintf(stderr, "␣bad␣link␣#:␣%d\n", i);
        break;
      }
      new_agree_link(S−>L + i, k);
      break;

    case 's':
    case 'x':
      WriteSolns( );
      break;

    case 'v':
      WriteLinkVars(S);
      break;

    case 'y':
      WriteLinkEqs(S);
```

```
    break;

  case 'g':
    WriteAgreedLinks(S);
    break;

  case 'r':
  case 'l':
    n = getparam( input, SPECIAL );
    if (n == SPECIAL) {     /* do all */
      n = S->neqs;
      for (i = 0; i < n; ++i)
        eqlist[i] = i;
    }
    else if (n >= 0 && n <= S->neqs)
      for (i = 0; i < n; ++i) {
        if (scanf("%d", &j) < 1 || j < 0 || j >= S->neqs) {
          fprintf(stderr, " bad eq #: %d\n", j);
          n = i;
          break;
        }
        eqlist[i] = j;
      }
    else {
      fprintf(stderr, " bad # eqs: %d\n", n);
      break;
    }
    if (c == 'r') {
      i = ReduceEqs(S->E, n, eqlist);
      printf(" common dimension = %d\n", i);
    }
    if (c == 'l') {
      j = S->nlinks;
      i = LinkEqs(S, n, eqlist);
```

```
      if (i)
        printf("␣%2d␣new␣links,␣total␣dimension␣=␣%d\n", S->nlinks - j, i);
      else
        printf("␣no␣new␣linkage\n");
    }
    break;

  case 'i':
    i = getparam( input, SPECIAL );
    if (i == SPECIAL)
      for (i = 0; i < S->neqs; ++i)
        EquationInfo(S, i);
    else if (i >= 0 && i < S->neqs)
      EquationInfo(S, i);
    else
      fprintf(stderr, "␣bad␣eq␣#:␣%d\n", i);
    break;

  case 'p':
    i = getparam( input, SPECIAL );
    if (i == SPECIAL) {
      printf("␣␣Equations␣0␣-␣%d:\n", S->neqs - 1);
      for (i = 0; i < S->neqs; ++i)
        WriteEquation(S->E + i);
    }
    else if (i >= 0 && i < S->neqs){
      printf("␣␣Equation␣%d:\n", i);
      WriteEquation(S->E + i);
      break;
    }
    else
      fprintf(stderr, "␣bad␣eq␣#:␣%d\n", i);
    break;
```

```
case 'e':
   printf("␣␣Linear␣Equation:\n");
   WriteEquation(S->LE);
   break;

case 'k':
   i = getparam( input, SPECIAL );
   if (i == SPECIAL)
      for (i = 0; i < S->nlinks; ++i)
         WriteLink(S->L + i);
   else if (i >= 0 && i < S->nlinks)
      WriteLink(S->L + i);
   else
      fprintf(stderr, "␣bad␣link␣#:␣%d\n", i);
   break;

case 'z':
   i = getparam( input, 0 );
   if (i >= 0 && i < S->nlinks)
      WriteZ(S->L + i);
   else
      fprintf(stderr, "␣bad␣link␣#:␣%d\n", i);
   break;

case 'd':    /* rhs_disagree */
   i = getparam( input, 0 );
   if (i < 0 || i >= S->nlinks) {
      fprintf(stderr, "␣bad␣link␣#:␣%d\n", i);
      break;
   }
   n = S->L[i].neqs;
   for (k = 0; k < n; ++k) {
      if (scanf("%d", &j) < 1 || j < 0 || j >= S->E[S->L[i].eqlist[k]].nrhs) {
         fprintf(stderr, "␣bad␣RHS␣#␣for␣eq␣%d:␣%d\n", S->L[i].eqlist[k], j);
```

```
        break;
      }
      eqlist[k] = j;
    }
    if (k == n)
      printf("␣␣in␣link␣%d,␣given␣RHS␣%sagree\n", i, rhs_disagree(S->L + i, eqlist)
          ? "dis" : "");
    break;

case 'f':    /* find_rhs(Link * link, int zeq, int
                                                  * *irhs, int *out) */
    i = getparam( input, 0 );
    if (i < 0 || i >= S->nlinks) {
      fprintf(stderr, "␣bad␣link␣#:␣%d\n", i);
      break;
    }
    if (scanf("%d", &n) < 1) {
      fprintf(stderr, "␣missing␣#\n");
      break;
    }
    if (n < 0 || n >= S->neqs) {
      fprintf(stderr, "␣bad␣eq␣#:␣%d\n", n);
      break;
    }
    for (k = 0; k < S->L[i].neqs - 1; ++k) {
      if (scanf("%d", &j) < 1 || j < 0 || j >= S->E[S->L[i].eqlist[k]].nrhs) {
        fprintf(stderr, "␣bad␣RHS␣#␣for␣eq␣%d:␣%d\n", S->L[i].eqlist[k], j);
        break;
      }
      eqlist[k] = j;
    }
    if (k == S->L[i].neqs - 1) {
      j = find_rhs(S->L + i, n, eqlist, eqlist);
```

```
      printf("␣␣in␣link␣%d,␣given␣RHS␣for␣other␣eqs,␣in␣eq␣%d␣these␣%d␣RHS
          ␣agree:\n␣␣␣", i, n, j);
      for (k = 0; k < j; ++k)
        printf("␣%d", eqlist[k]);
      printf("\n");
    }
    break;

case 'w':
    WriteSystem(S);
    break;

case '*':
    i = 0;
    j = 1;
    scanf(InFormat, &i);
    i &= FieldMask;
    scanf(InFormat, &j);
    j &= FieldMask;
    printf("**␣multiply:␣␣");
    printf(OutFormat, i);
    printf("␣*␣");
    printf(OutFormat, j);
    printf("␣=␣");
    printf(OutFormat, fmul(i, j));
    printf("\n");
    break;

case '/':
    i = 0;
    j = 1;
    scanf(InFormat, &i);
    i &= FieldMask;
    scanf(InFormat, &j);
```

```
    j &= FieldMask;
    printf("** divide:  ");
    printf(OutFormat, i);
    printf(" / ");
    printf(OutFormat, j);
    printf(" = ");
    printf(OutFormat, fdiv(i, j));
    printf("\n");
    break;

  case '^':
    i = 0;
    j = 1;
    scanf(InFormat, &i);
    i &= FieldMask;
    scanf("%d", &j);
    printf("** power:  ");
    printf(OutFormat, i);
    printf(" ^ %d = ", j);
    printf(OutFormat, fpow(i, j));
    printf("\n");
    break;

  case '-':
    j = 1;
    scanf(InFormat, &j);
    j &= FieldMask;
    printf("** invert:  ");
    printf(OutFormat, j);
    printf(" ^{-1} = ");
    printf(OutFormat, finv(j));
    printf("\n");
    break;
```

```
        case '+':
          i = 0;
          j = 1;
          scanf(InFormat, &i);
          i &= FieldMask;
          scanf(InFormat, &j);
          j &= FieldMask;
          printf("** add:  ");
          printf(OutFormat, i);
          printf(" + ");
          printf(OutFormat, j);
          printf(" = ");
          printf(OutFormat, (i ^ j));
          printf("\n");
          break;

        case 't':    /* for testing */
          new_reinit_agreed( );
          break;

        case 'q':
          return 0;

        default:
          printf("%s",menu);
          break;
      }
      if (rv == Inconsistent)
        printf("Warning: system INCONSISTENT!\n");
        printf("> ");
    }

    return 0;
}
```

```
/*
        fieldarith.h
        version: 2013 Jan 22
** AES block cipher version
 representation of fields:
 "A" poly basis in 2^8 : A^8+A^4+A^3+A+1 = 0
 "alpha" poly basis in 2^4 : a^4+a+1 = 0
 "Omega" poly basis in 2^2 : w^2+w+1 = 0
 */

/* field multiplication done through lookup logs, add logs, lookup antilog
to avoid doing modulo (q-1) the antilog table is doubled
to avoid testing for zero, log(0) = infinity (really 2q-1)
so need to pad end of antilog table with zeros up to 3q-3, also at 4q-2
here, rounded up size of tables for alignment reasons
Note: below embed smaller antilog tables in unused portion of bigger ones
Warning: in macros below, inversion or division by zero gives wrong answer,
as does zero to a power
*/

#define fmul(x,y) (ALog[ Log[x]+Log[y] ]) /* field x*y */
#define finv( x ) (ALog[ FieldMask-Log[x] ]) /* field 1/x (x != 0) */
#define fdiv(x,y) (ALog[ Log[x]+FieldMask-Log[y] ]) /* field x/y (y != 0) */
#define fpow(x,y) (ALog[ (Log[x]*(y)) % FieldMask +FieldMask]) /* (field x)^(integer y) */

const Elem *ALog;
const int *Log;

const Elem ALogs[1024] = {
0x01,0x03,0x05,0x0F,0x11,0x33,0x55,0xFF,0x1A,0x2E,0x72,0x96,0xA1,0xF8,0x13,0x35,
0x5F,0xE1,0x38,0x48,0xD8,0x73,0x95,0xA4,0xF7,0x02,0x06,0x0A,0x1E,0x22,0x66,0xAA,
0xE5,0x34,0x5C,0xE4,0x37,0x59,0xEB,0x26,0x6A,0xBE,0xD9,0x70,0x90,0xAB,0xE6,0x31,
```

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

```
0x9B,0xB6,0xC1,0x58,0xE8,0x23,0x65,0xAF,0xEA,0x25,0x6F,0xB1,0xC8,0x43,0xC5,0x54,
0xFC,0x1F,0x21,0x63,0xA5,0xF4,0x07,0x09,0x1B,0x2D,0x77,0x99,0xB0,0xCB,0x46,0xCA,
0x45,0xCF,0x4A,0xDE,0x79,0x8B,0x86,0x91,0xA8,0xE3,0x3E,0x42,0xC6,0x51,0xF3,0x0E,
0x12,0x36,0x5A,0xEE,0x29,0x7B,0x8D,0x8C,0x8F,0x8A,0x85,0x94,0xA7,0xF2,0x0D,0x17,
0x39,0x4B,0xDD,0x7C,0x84,0x97,0xA2,0xFD,0x1C,0x24,0x6C,0xB4,0xC7,0x52,0xF6, 0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9, /* ALog4 = ALogs+3*256 */
1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9, 0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
1, 2, 3, /* ALog2 = ALogs+3*256+3*16 */
1, 2, 3, 0,0,
0,0,0,0,
1, /* ALog1 = ALogs+3*256+3*16+3*4 */
1, 0,0,
/*0,0,0,0, */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
};

const int Log8[256] = {
511, 0x00,0x19,0x01,0x32,0x02,0x1A,0xC6,0x4B,0xC7,0x1B,0x68,0x33,0xEE,0xDF,0x03,
0x64,0x04,0xE0,0x0E,0x34,0x8D,0x81,0xEF,0x4C,0x71,0x08,0xC8,0xF8,0x69,0x1C,0xC1,
```

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

const int Log4[16] = {
31, 0, 1, 4, 2, 8, 5, 10, 3, 14, 9, 7, 6, 13, 11, 12,
};

const int Log2[4] = {
7, 0, 1, 2,
};

const int Log1[2] = {
3, 0,
};
```

```
/****************** New Agreeing *****************/
/*
  version: 2013 Mar 22
  Each call of new_agree_link introduces one more link into the Agreeing struct
  if feasible, the number of consistent sets is calculated, without actually finding them
    this works if each new link includes a new eq with the rhss giving all possible z values
      (call this a "cover" eq since it covers the possibilities)
      (the links may need to be re-ordered to achieve this condition for all)
    then for any set of rhss of previous eqs, there is a rhs in the cover eq to sat the link
      [note: the count assumes exactly one rhs per z val in the cover eq.]
  if not, all consistent sets are found and counted (but not stored); call this MultiAgree
    (this is computationally intensive but does not require lots of memory)
    then if a rhs is not in any consistent set, it gets eliminated
  except: a MultiAgree may be suppressed by a flag parameter

  After the first MultiAgree, those links are called "agreed";
    newly added links with a new cover eq will still allow the count without finding sets
  */

//#define SHOWORDER // to see more details of process
#define NOSORT // can choose to skip sorting since there are n! permutations of n links

/* AgrLinkInfo is structure for info about a multilink */
typedef struct {
   int neqs;     /* number of eqs in link */
   int firstnew;    /* index of first cover eq */
   int eqend;    /* index into Agr.eqlist beyond last eq */
   int *eqlist;    /* sorted array of indices into Sys.E */
   int *leqlist;    /* parallel array of indices into Link.eqlist */
} AgrLinkInfo;

/* Agreeing is structure for agreed multilinks */
typedef struct {
```

```
  int nlinks;   /* number of agreed links */
  int neqs;    /* number of agreed MRHS equations */
  SetCtr nsets;   /* estimated number of consistent sets */
  int nagreed;   /* number of links actually multi-agreed */
  SetCtr nagreedsets;  /* number of sets actually multi-agreed */
  int NotAgreed;   /* skipped last multi-agree! */
  int maxlinks;   /* max number of links allocated */
  int maxnrhs;   /* max number of rhs of eqs */
  int *linklist;   /* array of indices into Sys.L for links */
  int *eqlist;   /* array of indices into Sys.E for eqs */
  int *eqidx;   /* array of indices from Sys.E into Agr.eqlist */
  AgrLinkInfo **linfo;   /* array of ptr->info for ea link */
} Agreeing;

/****************** Global Variables ****************/
Agreeing Agr;    /* global var */
void *gptr;    /* global pointer for temporary uses
                                                    * (agree_link needs one) */

int
CompareIdxs(const void *x, const void *y)
{
  /* for sorting or searching array of indices into list of z values: gptr */
  Elem a, b, *z;
  z = (Elem *) gptr;
  a = z[*((int *) x)];
  b = z[*((int *) y)];
  return (a < b) ? -1 : (a > b) ? 1 : 0;
}

int
CompareIndices(const void *x, const void *y)
{
  /* for sorting or searching array of indices into list (gptr) of integer values */
```

```
  int a, b, *z;
  z = (int *) gptr;
  a = z[*((int *) x)];
  b = z[*((int *) y)];
  return (a < b) ? -1 : (a > b) ? 1 : 0;
}

/* init_agreed initializes the Agreeing structure */
void
new_init_agreed(void)
{
  int i;

  Agr.nlinks = Agr.neqs = Agr.nagreed = Agr.maxnrhs = Agr.NotAgreed = 0;
  Agr.nsets = 0;
  Agr.nagreedsets = 1;
  Agr.maxlinks = S->maxlinks;
  Agr.linklist = (int *) malloc(S->maxlinks * sizeof(int)); /* room for all */
  Agr.eqlist = (int *) malloc(S->neqs * sizeof(int)); /* room for all */
  Agr.eqidx = (int *) malloc(S->neqs * sizeof(int)); /* room for all */
  Agr.linfo = (AgrLinkInfo **) malloc(S->maxlinks * sizeof(AgrLinkInfo*)); /* none to
      start */
  for(i=0;i<S->neqs;i++) Agr.eqidx[i]=-1; /* none to start */
  return ;
}

/* reinit_agreed reinitializes the Agreeing structure */
void
new_reinit_agreed(void)
{
  int i;

  for(i= Agr.nlinks-1; i>=0; i--) {
    free(Agr.linfo[i]->eqlist);
```

```
    free(Agr.linfo[i]);
  }
  for(i=0;i<S->neqs;i++) Agr.eqidx[i]=-1; /* none to start */
  Agr.nlinks = Agr.neqs = Agr.nagreed = Agr.maxnrhs = 0;
  Agr.nsets = 0;
  Agr.nagreedsets = 1;
  return ;
}

/*
 nextperm generates next permutation of list l[n] of integers
 assume start with l in increasing order
 returns index to first changed position
 after last permutation, returns -1, and l is back in incr order
 (Note: got this algorithm from Wikipedia, sez can handle repeated values)
 */
int nextperm(int n, int l[])
{
  int i,j,k,t;
  for (j=n-2; j >= 0 && l[j] >= l[j+1]; j--); // 1st change posn
  if (j>=0) { // not last permutation
    for (k=n-1; l[j] >= l[k]; k--); // smallest larger replacement
    t=l[j]; l[j]=l[k]; l[k]=t; // swap
  }
  for (i=1; j+i < n-i; i++) // reverse rest of list
  {t=l[j+i]; l[j+i]=l[n-i]; l[n-i]=t;} // swap
  return j; // -1 if was already last permutation; back in order
}

/*
 To agree a new link:
 if at least one new eq in new link, OK, can estimate # sets
 if no new, try to re-order links so each has new eq
 then if reordering works, OK
```

```
  if ordering fails, then need to find all consistent sets for accurate count
  */

/* new_agree_link incorporates new link into agreeing struct */
/* NOTE: ONLY WORKS IF LINK HAS DIMENSION 1 (like old "small link")
  BUT: with ability to SkipAgree, can effect higher dimension 1 row at a time
  SO: no longer need to even consider links of higher dimension */
Status // OK, Fail, Inconsistent, Changed
new_agree_link(Link * link, int SkipAgree) // SkipAgree suppresses Multi-Agree link
{
   int i, j, k, n, t, ieq, eqn, linkn;
   int nlinks2chk, neqs2chk ;
   int nnew, reordered=0;
   int *linkperm, *eqperm, *peqend, *peqidx, *l, *ll;
   AgrLinkInfo **li, *lip;
   Status stat = OK;

   linkn = link - S->L;
   // check here if already agreed
   for (i=0; i<Agr.nlinks; i++) if (Agr.linklist[i]==linkn) return OK;
   // info for new link. n link idx; k #eqs; j idx to eqlist; l eqlist; ll leqlist
   k = link->neqs;
   n = Agr.nlinks;
   Agr.linklist[n] = linkn;
   lip = Agr.linfo[n] = (AgrLinkInfo *) malloc(sizeof(AgrLinkInfo)); /* new info struct */
   l = lip->eqlist = (int *) malloc(k*2 * sizeof(int));
   ll = lip->leqlist = lip->eqlist + k;
// scan for new eqs in 2 stages, to store new non-cover eqs 1st, cover last
   j = Agr.neqs; // to index new eqs
   for ( nnew=ieq=0; ieq < k; ieq++ ) {
      eqn = link->eqlist[ieq];   /* eq number */
      i = Agr.eqidx[eqn]; // index
      if ( i < 0) { // then new (-1 is flag)
         nnew++; // count new
```

```
      if ( link->Z[ieq][0][S->E[eqn].nrhs] < FieldMask ) { // if non-cover
        i = j++; // save new index (do covers later)
        Agr.eqidx[eqn] = i;
        Agr.eqlist[i] = eqn ; // new eq in list
      }
      if (Agr.maxnrhs < S->E[eqn].nrhs) Agr.maxnrhs = S->E[eqn].nrhs; // track max
    }
    l[ieq] = i; // save index to Agr.eqlist
    ll[ieq] = ieq; // save index to L.eq
  }
  for ( ieq=0; ieq < k; ieq++ ) { // scan again for any new, cover eqs
    eqn = link->eqlist[ieq];   /* eq number */
    i = Agr.eqidx[eqn]; // index
    if ( i < 0) { // then still new (cover)
      i = j++; // save new index
      Agr.eqidx[eqn] = i;
      Agr.eqlist[i] = eqn ; // new eq in list
      l[ieq] = i; // save index to Agr.eqlist
    }
  }
  // sort link eqs to Agr.eqlist order
  // note: here leqidx is 0,1,..(k-1), so gives index into eqidx, so sort works
  gptr = l; /* eqidx is array of indices for qsort of leqidx */
  qsort((void *) ll, (size_t) k, sizeof(int), CompareIndices); /* sort indices */
  for ( j=0; j < k; j++ ) // use leqidx to reorder eqidx
    l[j] = Agr.eqidx[ link->eqlist[ll[j]] ];
  lip->neqs = k;
  lip->eqend = l[k-1]+1; // beyond last eq
  lip->firstnew = k-nnew;
  Agr.neqs += nnew;
  Agr.nlinks++;
#ifdef SHOWORDER
  fprintf( stderr, "␣link␣#%d;␣eqs␣(%d␣new):", linkn, nnew);
  for ( ieq=0; ieq < k; ieq++ )
```

```
      fprintf( stderr, "␣%d␣[%d]{%d}", Agr.eqlist[l[ieq]], l[ieq], ll[ieq]);
   fprintf( stderr, "\n");
#endif

   /*
    find out whether can just count sets without constructing them...
    don't need to construct provided:
    each successive link has new eq
    if new link has new eq, it is in order, OK!
    else may be way to re−order links that works; try all permutations if nec.
    start w current order, mostly works.
    seek spot to insert new link so it has new eq,
    then iterate: check order; new permutation; until works or none left
    */

   if(nnew) { // have new eq; OK
#ifdef SHOWORDER
      j = lip−>firstnew; ieq = l[j]; eqn = Agr.eqlist[ieq];
      fprintf( stderr, "␣new␣link␣in␣order;␣have␣new␣eq␣%d␣[%d](%d){%d}\n",eqn,ieq,j,ll
           [j]);
#endif
      if(Agr.nlinks==1) Agr.nsets=1;
      i = Agr.nlinks−1; // setup for count
      goto done;
   }

// at this point, have no new eq; setup to re−order
   /*
    linkperm: indices into linklist, in trial order ([−1]=−1)
    peqend: indexed as linklist; end of eq perm for link ([−1]=0)
    eqperm: indices into eqlist, in link order
    peqidx: indexed as eqlist; indices into eqperm (=−1 to flag new)
    make new permutation of link order (up to first link that didn't work)
    flag eqs as new (just flag those that were "old" last time)
```

```
    see if new cover for each link
    */
  n=Agr.nlinks;
  k=Agr.neqs;
  linkperm = ( (int *) malloc( ((n+1+k)*2) * sizeof(int)) ) +1;
  l = (int *) malloc( ((k>n)?k:n) * sizeof(int)); // max for list length
  li = (AgrLinkInfo **) malloc(n * sizeof(AgrLinkInfo*)); /* for copy */
  peqend = linkperm+n+1;
  eqperm = peqend+n;
  peqidx = eqperm+k;
  linkperm[-1]=-1; // for convenience
  peqend[-1]=0; // for convenience
  for (i=0;i<n;i++) {linkperm[i]=i; peqend[i]=Agr.linfo[i]->eqend;}
  for (i=0;i<k;i++) eqperm[i]=peqidx[i]=i;

  if( Agr.nagreed && Agr.linfo[Agr.nlinks-1]->eqend <= Agr.linfo[Agr.nagreed-1]->
      eqend ) {
    // cannot estimate count: new link has only previously agreed eqs
    reordered = 0; // re-order won't work
#ifdef SHOWORDER
    fprintf( stderr, "␣new␣link␣has␣no␣unagreed␣eq;␣skip␣resort\n");
#endif
    goto MultiAgree; // to prepare to agree
  }

#ifndef NOSORT // can choose to skip sorting since there are n! permutations of n links
  {
    SetCtr didperms = 0; // diagnostic info
    linkperm += Agr.nagreed; // skip over previously agreed links
    n -= Agr.nagreed;
    reordered = 1;
    eqn = Agr.neqs; // number of "old" eqs
    i=n-1; // i+1 is number of indices to permute
    do {
```

```
        k = nextperm(++i,linkperm); // k idx of first change
        didperms++;
        if ( k < 0 ) {reordered = 0;break;} // if nothing worked
        // check if this order works
        t = eqn; // prev end of old eqs
        eqn = peqend[linkperm[k−1]]; // beyond here unsorted
        for (i= eqn; i< t; i++) peqidx[eqperm[i]]=−1; // flag as "new"
        for (i= k; i< n; i++) { // check each re−ordered link
          linkn = linkperm[i];
          for ( j=t=nnew=0; j<Agr.linfo[linkn]−>neqs; j++ ) { // each eq
            ieq = Agr.linfo[linkn]−>eqlist[j];
            if ( peqidx[ieq] < 0 ) { // if new
              t = peqidx[ieq] = eqn; // add to end of list
              eqperm[eqn++] = ieq;
              nnew++;
            }
            else if ( t < peqidx[ieq] ) t = peqidx[ieq]; // track end
          }
          if (!nnew) break; // no new cover for i, next permutation
          peqend[linkn] = t+1;
        }
      } while (i < n);
#if 1
      fprintf( stderr,
      "␣new␣link␣has␣no␣new␣eq;␣try␣RESORT:␣%lld␣permutations␣of␣%d␣links,␣%s\n
          ",
      didperms, n, reordered ? "WORKED" : "DID␣NOT␣WORK");
#endif
      linkperm −= Agr.nagreed; // shift back to include agreed links
      n += Agr.nagreed; // so now nlinks
    }
#endif

    // done seeking new order
```

```
  if (reordered) {
    // here, found order that works; clean up...
#ifndef NOSORT // can choose to skip sorting, including this re−order of eqs
/*
 this would be the place to put new cover last if possible
 for each reordered link, seek new eq with max # z vals,
 then rotate eqperm accordingly to put that eq last for indexing
 */
    for ( i=Agr.nagreed; i<n; i++ ){ // for each reordered link, in new order
      linkn = Agr.linklist[linkperm[i]];
      lip = Agr.linfo[linkperm[i]];
      for ( ieq = peqend[linkperm[i]]−1, t = 0, j = 0; j < lip−>neqs; j++ )
        if ( peqidx[lip−>eqlist[j]] >= peqend[linkperm[i−1]] && // if new and more z vals
             t <= S−>L[linkn].Z[lip−>leqlist[j]][0][S−>E[Agr.eqlist[lip−>eqlist[j]]].nrhs] )
                {
          t = S−>L[linkn].Z[lip−>leqlist[j]][0][S−>E[Agr.eqlist[lip−>eqlist[j]]].nrhs];
          ieq = peqidx[lip−>eqlist[j]]; // save idx into eqperm of max z vals
        }
      t = eqperm[ieq];
      for (j = ieq; j < peqend[linkperm[i]]−1; j++) { // rotate new eqs
        eqperm[j] = eqperm[j+1];
        peqidx[eqperm[j]] = j;
      }
      eqperm[j] = t; // put eq w/ min z vals last
      peqidx[eqperm[j]] = j;
    }
#endif
    // at this point, install current re−ordering, update set count
    k = Agr.nagreed;
    for ( i=k; i<n; i++ ){ // copy links; permute eq #s
      l[i] = Agr.linklist[i];
      lip = li[i] = Agr.linfo[i];
      lip−>eqend = peqend[i]; // new end
      for ( j=0; j< lip−>neqs; j++ )
```

```
            lip->eqlist[j] = peqidx[lip->eqlist[j]]; // new index for eq
        }
        for ( i=k; i<n; i++ ){ // permute links
          Agr.linklist[i] = l[linkperm[i]];
          Agr.linfo[i] = li[linkperm[i]];
        }
        k = peqend[Agr.nagreed - 1];
        for ( i=k; i<Agr.neqs; i++ ){ // copy eqs; update idx
          eqn = l[i] = Agr.eqlist[i];
          Agr.eqidx[eqn] = peqidx[i]; // new index
        }
        for ( i=k; i<Agr.neqs; i++ ) // permute eqs
          Agr.eqlist[i] = l[eqperm[i]];
// this would be a good point to sort the eqlist of each link!!!
        // now sort eq list of each link
        gptr = l; /* make local array globally available for qsort */
        for ( i=Agr.nagreed; i<n; i++ ){ // sort eq idx lists of resorted links
          lip = Agr.linfo[i];
          k = lip->neqs;
          for (j=0;j<k;j++) {
            eqperm[j]=j; // idx list to sort
            l[j]=lip->eqlist[j]; // copy
            peqidx[j]=lip->leqlist[j];
          }
          qsort((void *) eqperm, (size_t) k, sizeof(int), CompareIndices); /* sort indices */
          for (j=0;j<k;j++) {
            lip->eqlist[j]=l[eqperm[j]]; // sort lists from copies
            lip->leqlist[j]=peqidx[eqperm[j]];
          }
        }
        // at this point all is in order
      }

      else {
```

```
    /*
      here, prepare to agree:
      insert new link in order, based on current eqlist;
      agree only necessary links; may have others with new cover at end.
      order linfo->eqlist increasing within each link
      */
  MultiAgree:
    n = Agr.nlinks;
    l[0] = Agr.linklist[n-1]; // copy new link to insert
    li[0] = Agr.linfo[n-1];
    eqn = Agr.linfo[n-1]->eqend;
    for ( i=n-1; i>0 && Agr.linfo[i-1]->eqend > eqn; i-- ){ // find place for new link
      Agr.linklist[i] = Agr.linklist[i-1]; // shift by one
      Agr.linfo[i] = Agr.linfo[i-1];
    }
    Agr.linklist[i] = l[0]; // insert new link
    Agr.linfo[i] = li[0];
    nnew = i; // position of new link
#ifndef NOSORT // can choose to skip sorting; below may adjust order of new link among
    agreed
// check whether to put new link first
    if (i && Agr.linfo[i-1]->eqend == eqn) {
      for (j=i-1; j>0 && Agr.linfo[j-1]->eqend == eqn; j-- ) ; // find first w/ same end
      k = Agr.linfo[j]->eqend - ((j)? Agr.linfo[j-1]->eqend : 0); // # new eqs in link j
      n = li[0]->neqs;
      for (t=1; t <= n && li[0]->eqlist[n-t] == eqn-t; t++) ; t--; // # in sequence
      if (Agr.linfo[j]->eqend == eqn && k>t ) { // if j same end, more new
        for ( ; i>j; i-- ){ // put new link before j
          Agr.linklist[i] = Agr.linklist[i-1]; // shift by one
          Agr.linfo[i] = Agr.linfo[i-1];
        }
        Agr.linklist[i] = l[0]; // insert new link
        Agr.linfo[i] = li[0];
        nnew = i; // position of new link
```

```
/* at this point, old link i+1 has at least one eq "newer" than new link i
  * so want to re−establish correct eq order: newer after older */
// now need to renumber t eqs, reorder them
for (j=0;j<Agr.neqs;j++) eqperm[j]=peqidx[j]=j;
for (j=eqn−k; j < eqn; j++) peqidx[j]=−1; // mark new ones
for (i= 0, t=eqn−k; i< 2; i++) { // check both re−ordered links
   lip = Agr.linfo[nnew+i];
   for ( j=0; j < lip−>neqs; j++ ) { // each eq
      ieq = lip−>eqlist[j];
      if ( peqidx[ieq] < 0 ) { // if new
         peqidx[ieq] = t; // add to end of list
         eqperm[t++] = ieq;
         lip−>eqend = t; // mark end of list
      }
   }
} // now have set permutation of k eqs.
// at this point, install current re−ordering
for ( i=nnew; i<Agr.nlinks; i++ ){ // permute eq #s
   lip = Agr.linfo[i];
   for ( j=0; j< lip−>neqs; j++ )
      lip−>eqlist[j] = peqidx[lip−>eqlist[j]]; // new index for eq
}
for (j=eqn−k; j < eqn; j++){ // copy eqs; update idx
   ieq = l[j] = Agr.eqlist[j];
   Agr.eqidx[ieq] = peqidx[j]; // new index
}
for (j=eqn−k; j < eqn; j++) // permute eqs
   Agr.eqlist[j] = l[eqperm[j]];
// this would be a good point to sort the eqlist of each link to be agreed!!!
// now sort eq list of each link
gptr = l; /* make local array globally available for qsort */
for ( i=nnew; i<Agr.nlinks; i++ ){ // sort eq idx lists of resorted links
   lip = Agr.linfo[i];
   n = lip−>neqs;
```

```
            for (j=0;j<n;j++) {
              eqperm[j]=j; // idx list to sort
              l[j]=lip->eqlist[j]; // copy
              peqidx[j]=lip->leqlist[j];
            }
            qsort((void *) eqperm, (size_t) n, sizeof(int), CompareIndices); /* sort indices */
            for (j=0;j<n;j++) {
              lip->eqlist[j]=l[eqperm[j]]; // sort lists from copies
              lip->leqlist[j]=peqidx[eqperm[j]];
            }
          }
          // at this point all is in order
        }
      }
#endif
      // how many links to agree?
      if ( nnew <= Agr.nagreed ) { // new link inserted in agreed
        nlinks2chk = Agr.nagreed+1; // one more link for finding sets
        neqs2chk = Agr.linfo[Agr.nagreed]->eqend; // end of eqs to check for sets
      }
      else {
        nlinks2chk = nnew+1; // end of links for finding sets
        neqs2chk = Agr.linfo[nnew]->eqend; // end of eqs to check for sets
      }
    }
    free( li );
    free( l );
    free( linkperm - 1 );
#ifdef SHOWORDER
    fprintf(stderr,"␣new␣links:");
    for(i=0;i<Agr.nlinks;i++) { fprintf(stderr,"␣%X(",Agr.linklist[i]);
      for(j=0;j<Agr.linfo[i]->neqs;j++) fprintf(stderr,"%X",Agr.linfo[i]->eqlist[j]);
      fprintf(stderr,",%X)",Agr.linfo[i]->eqend); }
    fprintf(stderr,";␣eqs:");
```

```
  for(i=0;i<Agr.neqs;i++) fprintf(stderr,"␣%X[%X]",Agr.eqlist[i],Agr.eqidx[Agr.eqlist[i]]);
  fprintf(stderr,"\n");
#else
  fprintf(stderr,"␣current␣links:");
  for(i=0;i<Agr.nlinks;i++) fprintf(stderr,"␣%d",Agr.linklist[i]);
  fprintf(stderr,";␣eqs:");
  for(i=0;i<Agr.neqs;i++) fprintf(stderr,"␣%d",Agr.eqlist[i]);
  fprintf(stderr,"\n");
#endif

  if ( ! reordered ) {
    fprintf(stderr,"␣␣MultiAgree!␣\n");
    if (SkipAgree) { // suppress Multi−Agree until next time
      long long int g=1;
      fprintf(stderr, "␣No:␣Let's␣NOT␣and␣say␣we␣did!␣Guesstimate␣#␣of␣sets...\n");
      Agr.NotAgreed = 1;
      Agr.nagreed = nlinks2chk; // pretend agreed these links
      Agr.nagreedsets = g << (Nbits∗(neqs2chk−nlinks2chk)); // should have fewer sets
    } else
    /∗
     This is messy...
     The main idea is to check that each combination of rhs of eqs satisfies each link
     For each link that has new eqs, for last eq use z val to lookup rhs index
     call this indexed eq (other type called variable eq), distinguish 2 cases:
     if has some repeated z vals, may change idx satisfying link; track this
     else just look up idx to rhs
     Figure out next combo to try based on link that failed

     I use a lot of lookup tables, hoping that is more efficient than recalculating
     combo: list of indices of rhs for each eq (non−indexed eqs count down to 0)
     needrhs: for each rhs of each eq, flag whether gets used in consistent set (to elim eqs)
     eqlink: for each eq, index of first link that includes it
     changeq: for each link, idx of eq to change if link fails (for next combo)
```

```
    skipped: for each link, 0 or idx of last changeable eq (variable or rep−z) skipped by
        changeq
    preveq: for each eq, idx of previous eq to change (for next combo)
    Zp: for each link, pointer to array of pointers to Z lists, one for each eq in link
    idxd: for each eq, flag whether indexed (1, or −1 if repeated z vals) or not (0)
    znum: for each indexed eq, for each z, either: rhs idx, or −1 if none, or −n−1 if n
        repeated
    rhsidx: for each repeated−z link, list of indices to rhs, sorted by increasing z val
    zstart: for each repeated−z link, for each z, pointer into rhsidx to first for this z val
    zp: for each repeated−z link, list of rhs for current z val
    zrep: for each repeated−z link, remaining number of repeated current z vals
    */
   {
#include <time.h>
    clock_t cstart, cfinish;
    register int z, j, ae, il;
    int *eqs, *eqsend, *lastrhs, *skipped, ***zstart;
    Elem **Z, ***Zp;
    int **needrhs, *idxd;
    int **rhsidx, **znum, **lasteq, *combo;
    int *eqlink, *preveq, *changeq, *zrep, **zp, zcnt[Field];
    int row;
    Elem *zl;
    SetCtr didcombos, didlinks;

#ifdef SHOWORDER
    fprintf(stderr,"to␣check␣%d␣links␣of␣%d␣eqs\n",nlinks2chk,neqs2chk);
#endif
    // allocate & setup tables
    combo = (int *) malloc(neqs2chk * sizeof(int)); // test combo of rhs idx
    needrhs = (int **) malloc(neqs2chk * sizeof(int *));
    for (i=0;i<neqs2chk;i++)
      needrhs[i] = (int *) calloc(S−>E[Agr.eqlist[i]].nrhs, sizeof(int));
    znum = (int **) malloc((nlinks2chk*4) * sizeof(int *));
```

```
rhsidx = znum + nlinks2chk;
zp = rhsidx + nlinks2chk;
lasteq = zp + nlinks2chk;
eqlink = (int *) malloc((neqs2chk*4+nlinks2chk*3+1) * sizeof(int));
preveq = eqlink + neqs2chk;
idxd = preveq + neqs2chk;
lastrhs = idxd + neqs2chk;
skipped = lastrhs + neqs2chk;
zrep = skipped + nlinks2chk;
changeq = zrep + nlinks2chk; // changeq has one beyond end for convenience
Zp = (Elem ***) malloc(nlinks2chk * sizeof(Elem **));
for (n=il=0; il < nlinks2chk; il++) n += Agr.linfo[il]->neqs; // tot num of Z lists
Z = (Elem **) malloc(n * sizeof(Elem *));
zstart = (int ***) malloc((nlinks2chk) * sizeof(int **));
for (j = 0; j < nlinks2chk; j++) zstart[j] = NULL; // mostly not needed

ae = -1;
for (il=ieq=0; il < nlinks2chk; il++) { // relate eqs to links
  linkn = Agr.linklist[il];
  lip = Agr.linfo[il];
  n = lip->neqs; // for convenience
  lasteq[il] = lip->eqlist + n - 1 ; // ptr to penultimate eq, for convenience
  Zp[il] = Z;
  for (j=0; j<n; j++) {
    (*Z++) = S->L[linkn].Z[ lip->leqlist[j] ][0]; // list of z vals
  }
  znum[il] = NULL; // in case no index
  for (;ieq< Agr.linfo[il]->eqend;ieq++) { // each eq (count on ordering)
    eqn = Agr.eqlist[ieq];
    n = S->E[eqn].nrhs;
    lastrhs[ieq] = n-1; // idx to last
    eqlink[ieq] = il;
    preveq[ieq] = ae; // prev eq to change if cannot change this one
    if (ieq+1==Agr.linfo[il]->eqend) { // make index for last eq
```

```
        idxd[ieq] = 1; // flag as indexed
        zl = Z[−1]; // list of z vals
#ifdef SHOWORDER
        fprintf(stderr,"setup␣link␣[%d]=%d,␣eq␣[%d]=%d,␣%d␣rhs\n␣Z:",il,linkn,ieq,
            eqn,n);
        for(j = 0; j <= n; j++)
          fprintf(stderr,"␣%X",zl[j]);
        fprintf(stderr,"\n");
#endif
        if(n−1−zl[n]) { // if more rhs than z vals, some zs must repeat
          ae = ieq; // this eq may be changeable when z repeats
          idxd[ieq] = −1; // flag as indexed w/ rep
          zrep[il] = 0;
          znum[il] = (int ∗) malloc((Field + n) ∗ sizeof(int));
          rhsidx[il] = znum[il] + Field;
          for (j=0;j<Field;j++) zcnt[j] = 0; // zero count
          for (j=0;j<n;j++) {
            zcnt[zl[j]]++; // count z vals
            rhsidx[il][j] = j; // initialize idx list
          }
          gptr = zl; // global ptr for qsort
          qsort((void ∗) rhsidx[il], (size_t) n, sizeof(int), CompareIdxs); /∗ sort indices
              ∗/
          zstart[il] = (int ∗∗) malloc((Field) ∗ sizeof(int ∗));
          zstart[il][0] = rhsidx[il];
          for (j = 0; j < FieldMask; j++)   /∗ accumulate #zs to idx rhsidx∗/
            zstart[il][j+1] = zstart[il][j] + zcnt[j];
          for (j=0;j<Field;j++)
            znum[il][j] = ( zcnt[j] == 1 ) ? zstart[il][j][0] : −(zcnt[j]+1);
#ifdef SHOWORDER
          fprintf(stderr,"zn:");
          for(j = 0; j < Field; j++)
            fprintf(stderr,"␣%d",znum[il][j]);
// fprintf(stderr,"; zs:");
```

```
// for(j = 0; j < Field; j++)
// fprintf(stderr," %d",(int)(zstart[il][j]−rhsidx[il]));
                fprintf(stderr,";␣ri:");
                for(j = 0; j < n; j++)
                  fprintf(stderr,"␣%d",rhsidx[il][j]);
                fprintf(stderr,"\n");
#endif
              }
              else {
                znum[il] = (int ∗) malloc((Field) ∗ sizeof(int));
                for (j = 0; j < Field; j++) znum[il][j] = −1; // default sez none
                for (j=0;j<n;j++)
                  znum[il][zl[j]] = j; // rhs idx for that z val
#ifdef SHOWORDER
                fprintf(stderr,"zn:");
                for(j = 0; j < Field; j++)
                  fprintf(stderr,"␣%d",znum[il][j]);
                fprintf(stderr,"\n");
#endif
              }
            } else {
              ae = ieq; // this is a "variable" eq, not indexed
              idxd[ieq] = 0;
            }
          }
/∗ seek changeq: eq to change if link fails (latest changeable eq that affects link)
   k is a changeable eq that would affect link; i is eq to affect in link ∗/
        n = lip−>neqs; // for convenience
        for (k=0, j= (eqlink[ieq−1]==il) ? n−1 : n; j; ) { // skip last eq in idxd link
          i = lip−>eqlist[−−j]; // idx of eq ( work backwards )
          if (i <= k) break; // found max k
          if (idxd[i]<=0) { k = i; break; } // this changeable eq is k
          if ( k < changeq[eqlink[i]] ) k = changeq[eqlink[i]]; // current max
        }
```

```
      changeq[il] = k;
      i = (eqlink[ieq−1]==il && ieq−1==ae) ? preveq[ae] : ae; // max changeable
      skipped[il] = (i > k) ? i : 0; // note if skipped changeable eq
    }
    changeq[nlinks2chk] = ae; // beyond end, for convenience
#ifdef SHOWORDER
    fprintf(stderr,"changeq[sk]:");
    for(j = 0; j <= nlinks2chk; j++) {
      fprintf(stderr,"␣%d",changeq[j]);
      if (j < nlinks2chk && skipped[j]) fprintf(stderr,"[%d]",skipped[j]); }
    fprintf(stderr,";␣preveq:");
    for(j = 0; j < neqs2chk; j++)
      fprintf(stderr,"␣%d",preveq[j]);
    fprintf(stderr,"\n");
#endif

    /*
     The multiagree loop begins at checkcombo
     use "spaghetti code" (using goto) for efficiency
     */
    cstart = clock();
    for (ieq = changeq[nlinks2chk]; ieq >= 0; ieq = preveq[ieq]) // initialize combo
      combo[ieq] = lastrhs[ieq]; // idx to rhs for eq; count down from end
    didcombos = didlinks = 0;
    Agr.nsets = 0;
    ae = −1; // ae: combo agreed up to this eq idx
    il = 0; // il: idx of link to check
    // while (1) { // for ea combo [a combinatorially large number] (break at bottom)
    // loop back to here using goto, not loop structure
checkcombo: // come back here to try next combo
    didcombos++; // count combos for diagnostic
    for ( ; il < nlinks2chk; il++ ) { // check each unagreed link
      didlinks++; // count links checked for diagnostic
      Z = Zp[il]; // set up to sum z vals
```

```
            eqs = Agr.linfo[il]->eqlist;
            eqsend = lasteq[il];
            for (z = 0; eqs < eqsend; ) // for all but last eqn
               z ^= (*Z++)[combo[*eqs++]]; // add up zs
            eqn = *eqs;
            if ( ae < eqn ) { // if last eqn idxd (new)
               j = znum[il][z]; // look up index for this z
               if ( j >= 0 ) combo[eqn] = j; // valid rhs idx if not neg
               else if ( j == -1 ) goto checkskipped; // -1 flags no such z
               else {
                  zrep[il] = j = -(j+2); // neg indicates repeated z vals (offset)
                  zp[il] = zstart[il][z]; // array for this z
                  combo[eqn] = zp[il][j]; // idx from last rep
               }
            }
            else if ( z != (*Z)[combo[eqn]] ) goto checkskipped; // if no match, link fails
            ae = eqn; // combo agreed up to here so far
         }
         // at this point, have agreed set: count it; then go to next
         Agr.nsets++;
         for(j = 0; j < neqs2chk; j++)
            needrhs[j][combo[j]] = 1; // keep track of which rhs are in sets
         ieq = changeq[nlinks2chk]; // eq to change (last possible)
         goto nextcombo; // did not skip any
checkskipped: // come here when a link fails
         ieq = changeq[il]; // eq to change
         if ( (ae = skipped[il]) ) do { // if prev skipped over changes in combo, reset
            if (idxd[ae]) zrep[eqlink[ae]] = 0;
            else combo[ae] = lastrhs[ae];
            ae = preveq[ae];
         } while ( ae > ieq );
nextcombo:
         do { // find next combo, starting at ieq
            il = eqlink[ieq];
```

```
      if (idxd[ieq]) { // if indexed, then may be repeated zs
        if (zrep[il]) { // if repeats remain for same z
          combo[ieq] = zp[il][ −−zrep[il] ]; // try next
          il++; // this link OK, on to next
          ae = ieq;
          goto checkcombo;
        }
      }
      else // variable eq
        if (combo[ieq]−−) { // if haven't tried 'em all
          ae = ieq−1; // try this one
          goto checkcombo;
        }
        else // reset and back up
          combo[ieq] = lastrhs[ieq];
      ieq = preveq[ieq]; // back up to prev eq to change
    } while ( ieq >= 0 ); // while got another combo to check
    // break; // tried all combos: done
    // }
    Agr.nagreed = nlinks2chk; // now agreed these links
    Agr.nagreedsets = Agr.nsets; // now agreed these sets
    Agr.NotAgreed = 0; // yes, we did the MultiAgree
    cfinish = clock();
    cfinish = (cfinish > cstart) ? (cfinish−cstart) : (cfinish+(ULONG_MAX − cstart));
#if 1 //def SHOWORDER
    fprintf(stderr,"␣Checked␣%lld␣links␣in␣%lld␣combos␣of␣rhs.", didlinks, didcombos
        );
    fprintf(stderr,"␣(took␣%f␣seconds)\n", ((double)(cfinish))/ CLOCKS_PER_SEC );
#endif

    // all done finding sets. figure out whether can elim any rhs!
    ll = (int *) malloc(Agr.maxnrhs * sizeof(int)); // for rhs permutation
    for (ieq=0;ieq<neqs2chk;ieq++) { // each eq
      eqn = Agr.eqlist[ieq];
```

```
        n = S->E[eqn].nrhs;
#ifdef SHOWORDER
        fprintf(stderr," eq[%d] = %d w %d rhs; need:", ieq, eqn, n);
        for(j = 0; j < n; j++)
          needrhs[ieq][j] ? fprintf(stderr,"%2X",j) : fprintf(stderr," .");
        fprintf(stderr,"\n");
#endif
        for(j = n-1; j >= 0 && needrhs[ieq][j]; j--); // check if need all (find first not
            needed)
        if ( j >= 0 ) { // can elim some rhs
          // set up to remove rhs: use index permutation to reorder; for each link count z vals
          for( k = 0; k < n; k++) // identity permutation to start
            ll[k] = k;
          k--; // k is last rhs
          for( ; j >= 0; j--) // permute indices
            if (! needrhs[ieq][j] ) {
              if (j < k)
              { t = ll[j]; ll[j] = ll[k]; ll[k] = t; } // swap
              k--;
            }
          k++; // k is new # rhs
          fprintf(stderr,"! eq%3d can elim %3d rhs (keep %3d)!\n", eqn, n-k, k);
#ifdef SHOWORDER
          fprintf(stderr," new order:");
          for(j = 0; j < n; j++)
            fprintf(stderr," %X",ll[j]);
          fprintf(stderr,"\n");
#endif
#ifndef SAVERHS // can opt not to eliminate RHSs for experimental purposes
          stat = Changed;
          for( j = k; j < n; j++) // free rest
            free( S->E[eqn].B[ll[j]] );
          for( j = 0; j < k; j++) // permute rhs
            S->E[eqn].B[j] = S->E[eqn].B[ll[j]];
```

```
            S->E[eqn].nrhs = k;
            for( il = 0; il < S->E[eqn].nlinks; il++) { // permute Z arrays
              linkn = S->E[eqn].linklist[il];
              for ( i=0; i < S->L[linkn].neqs && S->L[linkn].eqlist[i] != eqn; i++); //
                  DANGER if corrupted!!
              if ( i == S->L[linkn].neqs ){
                fprintf(stderr,"DID␣NOT␣FIND␣EQ␣%d␣in␣link␣%d!\n", eqn, linkn);
                return Fail; }
              for ( row=0; row < S->L[linkn].dim; row++ ) {
                for( j = 0; j < Field; j++) zcnt[j]=0; // to count z
                for( t=j = 0; j < k; j++) { // permute Z row (in place)
                  z = S->L[linkn].Z[i][row][j] = S->L[linkn].Z[i][row][ll[j]];
                  if ( ! zcnt[z]++ ) t++; // count new z vals
                }
                S->L[linkn].Z[i][row][k] = t-1; // save # z vals (offset -1)
              }
            }
#endif
          }
        }

        // free up allocated spaces
        free( ll );
        for (i=nlinks2chk; i; )
        { free(zstart[--i]); free(znum[i]); }
        free( zstart );
        free( Zp[0] );
        free( Zp );
        free( eqlink );
        free( znum );
        for (i=neqs2chk; i; )
          free( needrhs[--i] );
        free( needrhs );
        free( combo ); // test combo of rhs idx
```

```
      }
    }
    // calc # sets from agreed and remaining new links
    Agr.nsets = Agr.nagreedsets;
    i=Agr.nagreed;
done: // jump here if new link has new eq, with i = nlinks − 1
    for ( ; i<Agr.nlinks; i++ ) {
      k = i ? Agr.linfo[i−1]−>eqend : 0; // 1st new eq
      lip = Agr.linfo[i];
      n = lip−>eqend − 1; // last new eq
      if ( n == k ) continue;
      for ( j=0; j < lip−>neqs; j++ ) { // for new eqs except last, mult by #rhs
        ieq=lip−>eqlist[j];
        if ( ieq >= k && ieq < n )
          Agr.nsets ∗= S−>E[Agr.eqlist[ieq]].nrhs; //

      }
    }
    printf("new:␣%3d␣links,␣agreed␣%d␣eqs␣for␣%lld␣sets%s\n", Agr.nlinks, Agr.neqs, Agr
        .nsets,
          Agr.NotAgreed ? "␣(approx.)" : "" );
    return stat;
}


/∗
  FindSets should only get called from WriteSolns,
  which should only get called after last MultiAgree
  so we assume eqs in Agr.eqlist are ordered to incr w/ links
  ∗/
void
FindSets(RhsIdx ∗∗C, int nsets)
{
    int i, j, il, eqn, leqn, agreed, chgeq;
```

```
int nlinks, neqs, sets;
int *combo;
Elem z;

neqs = Agr.neqs; nlinks = Agr.nlinks;
combo = (int *) malloc(neqs * sizeof(int)); // to test combo of rhs idx
/* ASSUME: just did final multi-agree, ALL links in order, ALL eqs in order */
for (i=0; i < neqs; i++) { // initialize combo
   eqn = Agr.eqlist[i];
   combo[i] = S->E[eqn].nrhs - 1; // count down from end
}
/*
 want eqs in link order so can tell first link to check
 il: idx of link to check
 */
sets = 0;
chgeq = -1; // all new combo
do { // for ea combo
   for (il = 0; il < nlinks && Agr.linfo[il]->eqend <= chgeq; il++ ); // skip OK links (no
        change)
   for ( agreed = 1; il < nlinks; il++ ) { // check affected links
      // consistency check, just add up zs
      for (z=j=0;j< Agr.linfo[il]->neqs; j++) { // add up zs
         eqn = Agr.linfo[il]->eqlist[j];
         leqn = Agr.linfo[il]->leqlist[j];
         z ^= S->L[ Agr.linklist[il] ].Z[leqn][0][combo[eqn]];
      }
      if (z) { // failed this link
         chgeq = eqn; // (last)
         agreed = 0; // flag no go
         break; // go to next combo
      }
   }
   // at this point, if agreed, save set; then go to next
```

```
    if ( agreed ) {
      if (sets >= nsets) {
        fprintf(stderr, "␣Found␣too␣many␣sets:␣%d␣>␣%d\n", ++sets, nsets);
      }
      else {
        for (i=0; i<neqs; i++)
          C[Agr.eqlist[i]][sets] = combo[i]; // C indexed like Sys.E
        sets++;
      }
      chgeq = neqs−1; // (last)
    }
    /*
     now need to figure out which combo to try next...
     if more rhs to try for current eq, try one
     else, reset this eq and back up to prev eq
     */
    for ( ; chgeq>=0 && !(combo[chgeq]−−); chgeq−−) // if back up, reset eqs
      combo[chgeq] = S−>E[Agr.eqlist[chgeq]].nrhs − 1;
  } while ( chgeq>=0 ); // while got another combo to check
  if (sets < nsets) {
    fprintf(stderr, "␣Found␣too␣few␣sets:␣%d␣<␣%d\n", sets, nsets);
  }
  free(combo);
}

int
JordanForm(Mat X, int nrows, int ncols)
{
  /* take row−reduced matrix of full rank with unit pivots, do back substitution */
  int r, c, p, pc, rv;
  const Elem *SclTab;

  for (p = nrows − 1; p > 0; p−−) {  /* sc is leading col of submatrix */
    /* find pivot */
```

```
    for (pc = p; pc < ncols; pc++)
      if (X[p][pc])
        break;    /* found pivot */
    if (pc == ncols) {    /* then done; rest is 0 */
      fprintf(stderr, "␣JordanForm␣matrix␣not␣full␣rank!\n");
      return −1;
    }
    if (p == nrows − 1)
      rv = pc;
    /* clear column */
    for (r = 0; r < p; r++)
      if (X[r][pc]) {
        SclTab = ALog + Log[X[r][pc]];
        for (c = pc; c < ncols; c++)
          X[r][c] ^= SclTab[Log[X[p][c]]]; /* subtract pivot row */
      }
  }
  return rv;
}

/*
 WriteSolns should be called only after final multi−agree.
 then it calls FindSets to determine consistent sets, put 'em in C matrix
 makes big matrix system, solves by Gauss−Jordan reduction
 */
void
WriteSolns(void)
{
  /* writes solutions from agreed sets */
  int i, eqn, rank, row, col, brow, nrows, ncols, cut, nsets;
  RhsIdx **C;    /* agreed rhs: list for each eq in S */
  Mat Aug;

  if (Agr.neqs != S−>neqs || Agr.nlinks != S−>nlinks) { /* not all agreed yet */
```

```
   fprintf(stderr, "␣Can't␣compute␣solutions␣until␣all␣eqs␣&␣links␣agreed!\n");
   return;
}
nsets = Agr.nsets; /* assume same as nagreedsets, small # */
if ( nsets <= 0) { /* no solutions */
   fprintf(stderr, "␣No␣(%d)␣sets␣agreed!\n", nsets);
   return;
}
// first need to find the sets again! Store them in C
/* allocate space for consistent sets C: 1 row per eq */
C = (RhsIdx **) malloc(S->neqs * sizeof(RhsIdx *)); /* none to start */
for (i = 0; i < S->neqs; i++)
   C[i] = (RhsIdx *) malloc(nsets * sizeof(RhsIdx));
FindSets(C,nsets); // recalculate consistent sets


for (i = nrows = 0; i < S->neqs; i++)
   nrows += S->E[i].nrows;
ncols = Nvar + nsets;
/* allocate Aug Matrix */
Aug = (Mat) malloc(nrows * sizeof(Vec));
for (i = 0; i < nrows; i++)
   Aug[i] = (Vec) malloc(ncols * sizeof(Elem));
/* copy lin eqs into Aug */
for (eqn = row = 0; eqn < S->neqs; eqn++)
   for (brow = 0; brow < S->E[eqn].nrows; row++, brow++) {
      for (col = 0; col < Nvar; col++)
         Aug[row][col] = S->E[eqn].A[brow][col];
      for (i = 0; i < Agr.nsets; i++, col++)
         Aug[row][col] = S->E[eqn].B[C[eqn][i]][brow];
   }

rank = ReduceMats(Aug, Aug, nrows, ncols, 0);
i = JordanForm(Aug, rank, ncols);
```

```
  if (i > Nvar) {    /* inconsistent! */
    fprintf(stderr, "␣Some␣set(s)␣inconsistent!\n");
    return;
  }
  if (rank < Nvar) {    /* free var! */
    fprintf(stderr, "␣Some␣var(s)␣free!\n");
    return;
  }
  /* now have all solutions */

  cut = ((Nbits > 4) ? 32 : 64) − 1;  /* line length = 64 for B */
  printf("␣Got␣%lld␣solution%s:\n", Agr.nsets, Agr.nsets > 1 ? "s" : "");
  /* write solns */
  for (i = 0; i < Agr.nsets; i++) {
    for (row = 0; row < Nvar; row++) {
      printf(OutFormat, Aug[row][Nvar + i]);
      if ((row & cut) == cut)
        printf("\n");
    }
    if (row & cut)
      printf("\n");
  }
  for (i = 0; i < nrows; i++)
    free(Aug[i]);
  free(Aug);
  for (i = 0; i < S−>neqs; i++)
    free( C[i] );
  free(C);
}

void
WriteAgreedLinks(Sys * S)
{
  /* writes CSV file of eqs in each link */
```

```
FILE *fout;
Link *link;
int i,j,k,c,il,ie,neqs;
int *linkcol;

/* which cols of U go with which eqs? */
neqs = Agr.nagreed ? Agr.linfo[Agr.nagreed-1]->eqend : 0;
linkcol = (int *) malloc((neqs + 1) * sizeof(int)); /* alloc linkcol */

fout = fopen("agreedlinks.csv", "w");
for (j=0; j< neqs; j++)
  fprintf(fout,",%d",Agr.eqlist[j]);
fprintf(fout, ",\n");
for (i=0; i<Agr.nagreed; i++) {
  il = Agr.linklist[i];
  fprintf(fout, "%d,", il);
  link = S->L + il;
  for (j = k = 0; j < link->neqs; j++) {
    linkcol[j] = k;
    k += S->E[link->eqlist[j]].nrows;
  }
  linkcol[j] = k;   /* end, for later convenience */
  for (k=j=0; j < neqs; j++) {
    if ( k < Agr.linfo[i]->neqs && j == Agr.linfo[i]->eqlist[k] ) {
      ie = Agr.linfo[i]->leqlist[k++];
      for (c = linkcol[ie]; c < linkcol[ie+1]; c++)
        fprintf(fout, OutFormat, link->U[0][c]);
    }
    fprintf(fout,",");
  }
  for (k=0; k < Agr.linfo[i]->neqs; k++)
    fprintf(fout," %d", Agr.eqlist[Agr.linfo[i]->eqlist[k]]);
  fprintf(fout, "\n");
}
```

```
  fclose(fout);
  printf("␣saved␣agreed␣link␣info␣to␣agreedlinks.csv\n");
}
```

newmrhs.h

```
/*
 Multiple Right−Hand Sides (MRHS) equations system solver
 version: 2013 Feb 26
 by D. Canright and. N Vanatta
 Starting from scratch, but heavily influenced by code of H. Raddum
 This one is for equations of field elements, not just bits. Each element fits in a byte.
 Code mostly not optimized for speed. (exception: multi−agree)
 */

/* #define SHOWMATRICES */
#define ONEDIMLINKS

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <math.h>

/* define # of ints needed for given # of bytes */
#if UINT_MAX == 255
#define INT_BITS 8
#define INT_BYTES 1
#define INT_BYTES_BITS 0
#elif UINT_MAX == 65535
#define INT_BITS 16
#define INT_BYTES 2
#define INT_BYTES_BITS 1
#elif UINT_MAX == 4294967295
#define INT_BITS 32
```

```
#define INT_BYTES 4
#define INT_BYTES_BITS 2
#elif UINT_MAX == 18446744073709551615
#define INT_BITS 64
#define INT_BYTES 8
#define INT_BYTES_BITS 3
#endif

#ifdef INT_BYTES_BITS
#define intlen(nbytes) (((nbytes) + INT_BYTES−1) >> INT_BYTES_BITS)
#else
#define intlen(nbytes) (((nbytes) + sizeof(int)−1)/sizeof(int))
#endif

/****************** Define Types ****************/
typedef unsigned char Elem;  /* field element */
typedef Elem *Vec;   /* vector */
typedef Vec *Mat;   /* matrix */

/* NOTE: trying to keep index in a byte; OK for AES if no flag value */
typedef unsigned char RhsIdx;  /* index to RHS in MRHS eq */
typedef unsigned long long SetCtr;  /* count large number of sets */

/* Eq is structure for MRHS equation */
typedef struct {
  int nrows;   /* number of linear equations in A */
  int nrhs;    /* number of RHS in B */
  int nlinks;   /* number of links involving this Eq */
  int maxlinks;   /* max number of links allocated to
                                                    * linklist */
  int *linklist;   /* list of link indices */
  Mat A;    /* linear part matrix */
  Mat B;    /* array of RHS vectors */
} Eq;
```

```
/* Link is structure for linear dependency among eqs */
typedef struct {
  int neqs;    /* number of MRHS equations in link */
  int dim;    /* number of rows in U */
  int ncols;   /* length of each row of U */
  int *eqlist;   /* array of indices to eqs */
  Mat U;    /* linear dependence matrix */
  Mat *Z;    /* matrix of z=Ub values for rhs b */
  /* NOTE: save of # of distinct z vals at end of each Z row */
} Link;

/* Sys is structure for system of MRHS equations */
typedef struct {
  int neqs;    /* number of MRHS equations in E */
  int nlinks;   /* number of links in L */
  int maxlinks;   /* max number of links allocated to L */
  Eq *E;    /* array of MRHS equations */
  Eq *LE;    /* pointer to linear equations for
                                                  * eliminated variables */
  Link *L;    /* array of links */
} Sys;

/* Status indicates possible results */
typedef enum {
  OK, Fail, Inconsistent, Changed  /* flags for results */
} Status;

/****************** Global Variables ****************/
int ARRAYLENGTH;
int Nvar, Nbits = 1, Field, FieldMask;
char *InFormat, *OutFormat;
Sys *S;
```

```
#ifdef SQUARE
#include "sqfieldarith.h" // log tables & macros
#else
#include "fieldarith.h" // log tables & macros
#endif

/****************** System Setup, I/O ****************/
void
NewEquation(Eq * eq, int nlinks)
{
  /* allocates memory for eq */
  int i;

  eq->A = (Mat) malloc(eq->nrows * sizeof(Vec));
  eq->B = (Mat) malloc(eq->nrhs * sizeof(Vec));
  for (i = 0; i < eq->nrows; ++i) {
    eq->A[i] = (Vec) malloc(Nvar * sizeof(Elem));
  }
  for (i = 0; i < eq->nrhs; ++i) {
    eq->B[i] = (Vec) malloc(eq->nrows * sizeof(Elem));
  }
  eq->linklist = (int *) malloc(nlinks * sizeof(int));
  eq->maxlinks = nlinks;
  eq->nlinks = 0;
}

void
NewLE(Eq * eq)
{
  /* sets up new LE matrix;
   allocates pointers for A, storage for B, enough to elim all vars */
  eq->nrows = 0;
  eq->nrhs = 1;
  eq->A = (Mat) malloc(Nvar * sizeof(Vec));
```

```
  eq->B = (Mat) malloc(sizeof(Vec));
  eq->B[0] = (Vec) malloc(Nvar * sizeof(Elem));
}

Status
NewLink(Sys * S, int neqs, int dim, int ncols, int *eqlist, Mat U)
{
  /* allocates (most) storage for link
   NOTE: calling routine must allocate & initialize eqlist & U before this call */
  void *p;
  Link *link;
  Eq *eq;
  const Elem *SclTab;
  int i, j, k, n, col, brow, more;
  int nz, newz[256]; // to tally distinct z values

  n = S->nlinks;
  if (n + 1 > S->maxlinks) {   /* need more room in L for links */
    more = S->maxlinks;   /* try for 100% more */
    while (more
           && (p = realloc((void *) S->L, (S->maxlinks + more) * sizeof(Link))) ==
              NULL)
      more >>= 1;
    if (!more)
      return Fail;
    S->L = (Link *) p;
    S->maxlinks += more;
  }
  link = S->L + n;
  link->neqs = neqs;
  link->dim = dim;
  link->ncols = ncols;
  link->eqlist = eqlist;
  link->U = U;
```

```
/* allocate Z matrix */
link->Z = (Mat *) malloc(neqs * sizeof(Mat));
if (link->Z == NULL)
   return Fail;
for (i = 0; i < neqs; i++) {
   eq = S->E + eqlist[i];
   link->Z[i] = (Mat) malloc(dim * sizeof(Vec));
   if (link->Z[i] == NULL)
      return Fail;
   for (j = 0; j < dim; j++) // one extra for count of z vals
      link->Z[i][j] = (Vec) calloc(eq->nrhs + 1, sizeof(Elem));
}
if (link->Z[i - 1][j - 1] == NULL)
   return Fail;

/* calculate Z matrix */
/* i: eq; j: row of U; col: col of U; brow: row of b; */
for (i = col = 0; i < neqs; i++) {
   eq = S->E + eqlist[i];
   for (brow = 0; brow < eq->nrows; brow++, col++)
      for (j = 0; j < dim; j++)
         if (U[j][col]) {
            SclTab = ALog + Log[U[j][col]];
            for (k = 0; k < eq->nrhs; k++)
               link->Z[i][j][k] ^= SclTab[Log[eq->B[k][brow]]]; /* add part */
         }
}

/* count distinct z values */
/* i: eq; j: row of U; k: rhs; */
for (i = 0; i < neqs; i++) {
   eq = S->E + eqlist[i];
   for (j = 0; j < dim; j++) {
      for (k = 0; k < Field; k++) newz[k] = 1;
```

```
        nz = −1; // bias to fit in char
        for (k = 0; k < eq−>nrhs; k++)
          if ( newz[link−>Z[i][j][k]] ) { nz++; newz[link−>Z[i][j][k]] = 0; } /* count distinct
              */
        link−>Z[i][j][k] = nz;
      }
    }

    for (i = 0; i < neqs; i++) {   /* inform eqs of link */
      eq = S−>E + eqlist[i];
      if (eq−>nlinks + 1 >= eq−>maxlinks) { /* need more room */
        more = eq−>maxlinks / 2;
        if ((p = realloc((void *) eq−>linklist, (eq−>maxlinks + more) * sizeof(int))) == NULL
            )
          return Fail;
        eq−>linklist = (int *) p;
        eq−>maxlinks += more;
      }
      eq−>linklist[eq−>nlinks++] = n;
    }
    S−>nlinks++;
    return OK;
}

void
NewSystem(Sys * S)
{
    S−>LE = (Eq *) malloc(sizeof(Eq));
    NewLE(S−>LE);
    S−>E = (Eq *) malloc(S−>neqs * sizeof(Eq));
    S−>nlinks = 0;
    S−>maxlinks = (S−>neqs * (S−>neqs − 1)) / 2; /* start with enough for all pairs */
    S−>L = (Link *) malloc(S−>maxlinks * sizeof(Link));
}
```

```
Status
ReadEquation(Eq * eq, FILE * fp)
{
  /* reads data for eq */
  int row, col;
  unsigned int num;

  /* read A by rows */
  for (row = 0; row < eq->nrows; row++) {
    for (col = 0; col < Nvar; col++) {
      if (fscanf(fp, InFormat, &num) != 1) {
        fprintf(stderr, "Error in readEquation: could not read A: row %d col %d\n",
            row, col);
        return Fail;
      }
      eq->A[row][col] = num & FieldMask;
    }
  }
  /* read b by cols */
  for (col = 0; col < eq->nrhs; col++) {
    for (row = 0; row < eq->nrows; row++) {
      if (fscanf(fp, InFormat, &num) != 1) {
        fprintf(stderr, "Error in readEquation: could not read B: col %d row %d\n",
            col, row);
        return Fail;
      }
      eq->B[col][row] = num & FieldMask;
    }
  }
  return 0;
}

/* ReadSystem reads in the system of equations; expected format:
```

```
 Nvar neqs nbits (all on one line; if nbits [# of bits] is missing, default is 1)
 nrows nrhs A B (for each Eq)
 */
#define LINELENGTH 128
Status
ReadSystem(Sys * S, char *filename)
{
  int i;
  FILE *fp;
  char line[LINELENGTH];

  fp = fopen(filename, "r");
  if (fp == NULL) {
    fprintf(stderr, "Error in ReadSystem: could not open file %s\n", filename);
    return Fail;
  }
  if (fgets(line, LINELENGTH - 2, fp) == NULL) {
    fprintf(stderr, "Error in ReadSystem: could not read header of file %s\n", filename
        );
    return Fail;
  }
  if (sscanf(line, "%d%d%d", &Nvar, &S->neqs, &Nbits) < 2) {
    fprintf(stderr, "Error in ReadSystem: could not read Nvar and neqs\n");
    return Fail;
  }
  if (Nvar < 1) {
    fprintf(stderr, "Error in ReadSystem: bad param: Nvar = %d\n", Nvar);
    return Fail;
  }
  if (S->neqs < 1) {
    fprintf(stderr, "Error in ReadSystem: bad param: S->neqs = %d\n", S->neqs);
    return Fail;
  }
  InFormat = "%1x";
```

```
OutFormat = "%01X";
switch (Nbits) {
  case 1:
    Log = Log1;
    ALog = ALogs + 3 * 256 + 3 * 16 + 3 * 4;
    Field = 2;
    break;
  case 2:
    Log = Log2;
    ALog = ALogs + 3 * 256 + 3 * 16;
    Field = 4;
    break;
  case 4:
    Log = Log4;
    ALog = ALogs + 3 * 256;
    Field = 16;
    break;
  case 8:
    Log = Log8;
    ALog = ALogs;
    Field = 256;
    InFormat = "%2x";
    OutFormat = "%02X";
    break;
  default:
    fprintf(stderr, "Error␣in␣ReadSystem:␣bad␣param:␣Nbits␣=␣%d\n", Nbits);
    return Fail;
}
FieldMask = Field − 1;

NewSystem(S);
for (i = 0; i < S−>neqs; ++i) {
  if (fscanf(fp, "%d%d", &S−>E[i].nrows, &S−>E[i].nrhs) != 2) {
```

```
        fprintf(stderr, "Error␣in␣ReadSystem:␣could␣not␣read␣nrows␣and␣nrhs␣for␣eq␣%d
            \n", i);
        return Fail;
      }
      if (S->E[i].nrows < 1) {
        fprintf(stderr, "Error␣in␣ReadSystem:␣bad␣param:␣S->E[%d].nrows␣=␣%d\n",
                i, S->E[i].nrows);
        return Fail;
      }
      if (S->E[i].nrhs < 1) {
        fprintf(stderr, "Error␣in␣ReadSystem:␣bad␣param:␣S->E[%d].nrhs␣=␣%d\n",
                i, S->E[i].nrhs);
        return Fail;
      }
      NewEquation(&S->E[i], S->neqs);
      if (ReadEquation(&S->E[i], fp)) {
        fprintf(stderr, "Error␣in␣ReadSystem:␣did␣not␣get␣eq␣%d\n", i);
        return Fail;
      }
    }
    return OK;
}

void
WriteMatrix(Mat M, int nrows, int ncols, char *tag)
{
    /* writes data for eq */
    int i, row, col, cut = -1;

    printf("␣%s[%dx%d]:\n", tag, nrows, ncols);
    /* write M by rows */
    for (row = i = 0; row < nrows; row++) {
      for (col = 0; col < ncols; col++) {
        printf(OutFormat, M[row][col]);
```

```
        if (!(++i & cut))
            printf("\n");
      }
      if (i & cut)
        printf("\n");
   }
}

void
WriteEquation(Eq * eq)
{
   /* writes data for eq */
   int i, row, col, cut;

   cut = ((Nbits > 4) ? 32 : 64) - 1;  /* line length = 64 for B */
   printf("␣%d␣%d\n", eq->nrows, eq->nrhs);
   /* write A by rows */
   for (row = 0; row < eq->nrows; row++) {
      for (col = 0; col < Nvar; col++) {
         printf(OutFormat, eq->A[row][col]);
      }
      printf("\n");
   }
   /* write B by cols */
   for (col = i = 0; col < eq->nrhs; col++) {
      for (row = 0; row < eq->nrows; row++) {
         printf(OutFormat, eq->B[col][row]);
         if (!(++i & cut))
            printf("\n");
      }
   }
   if (i & cut)
      printf("\n");
}
```

```
void
WriteLink(Link * link)
{
  /* writes data for link */
  int i, n;

  printf("␣link␣#");
  n = (int) (link - S->L);  /* link number? */
  if (n < 0 || n >= S->nlinks) {
       printf("␣%d␣??", n); return;}
  else
    printf("%3d", n);
  printf(";␣%d␣eqs:", link->neqs);
  for (i = 0; i < link->neqs; i++)
    printf("␣%d", link->eqlist[i]);
  printf(";");
  WriteMatrix(link->U, link->dim, link->ncols, "U");
}

void
WriteLinkVars(Sys * S)
{
  /* writes CSV file of vars in each link */
  FILE *fout;
  Link *link;
  Eq *eq;
  Vec linkvars;
  int i, j,k,r,c,flag;
  char *str[4] = { ",", ",.", ",x", ",*" };

  fout = fopen("linkvars.csv", "w");
  linkvars = (Vec) calloc(Nvar, sizeof(Elem));
  for (j=0; j<Nvar; j++) {
```

```
    fprintf(fout,",%d",j);
  }
  fprintf(fout, "\n");
  for (i=0; i<S->nlinks; i++) {
    fprintf(fout, "%d",i);
    link = S->L + i;
    for (c=k=0; k < link->neqs; k++) {
      eq = S->E + link->eqlist[k];
      for (r=0; r < eq->nrows; r++, c++) {
        flag = link->U[0][c] ? 2 : 1 ;
        for (j=0; j<Nvar; j++)
          if (eq->A[r][j]) linkvars[j] |= flag;
      }
    }
    for (j=0; j<Nvar; j++) {
      fprintf(fout,"%s",str[linkvars[j]]);
      linkvars[j] = 0;
    }
    fprintf(fout, "\n");
  }
  fclose(fout);
  printf("␣saved␣link␣variable␣info␣to␣linkvars.csv\n");
}

void
WriteLinkEqs(Sys * S)
{
  /* writes CSV file of eqs in each link */
  FILE *fout;
  Link *link;
  Eq *eq;
  int j,k,r,c,il,ie;

  fout = fopen("linkeqs.csv", "w");
```

```
  for (j=0; j< S->neqs; j++)
    fprintf(fout,",%d",j);
  fprintf(fout, ",\n");
  for (il=0; il<S->nlinks; il++) {
    fprintf(fout, "%d,",il);
    link = S->L + il;
    for (c=k=ie=0; ie < S->neqs; ie++) {
      if ( k < link->neqs && ie == link->eqlist[k]) {
        eq = S->E + link->eqlist[k++];
        for (r=0; r < eq->nrows; r++)
          fprintf(fout, OutFormat, link->U[0][c++]);
      }
      fprintf(fout,",");
    }
    for (k=0; k < link->neqs; k++)
      fprintf(fout," %d", link->eqlist[k]);
    fprintf(fout, "\n");
  }
  fclose(fout);
  printf(" saved link equation info to linkeqs.csv\n");
}

void
WriteSystem(Sys * S)
{
  /* writes data for system */
  int i;

  printf("  %d  %d  %d\n", Nvar, S->neqs, Nbits);
  /* write eqs */
  for (i = 0; i < S->neqs; i++)
    WriteEquation(S->E + i);
  if (S->LE->nrows) {
    printf(" linear eqs (%d):\n", S->LE->nrows);
```

```
      WriteEquation(S->LE);
    }
}

void
EquationInfo(Sys * S, int e)
{
  /* writes info for eq */
  int i;

  printf("Eq[%3d]:%3d rows,%3d rhs,%3d links",
          e, S->E[e].nrows, S->E[e].nrhs, S->E[e].nlinks);
  for (i = 0; i < S->E[e].nlinks; i++)
    printf(" %d", S->E[e].linklist[i]);
  printf("\n");
}

/****************** Solution Tools *****************/

int
ReduceMats(Mat X, Mat Y, int nrows, int xcols, int ycols)
{
  /* row reduce matrix X, same row ops on Y (if ycols != 0), return rank */
  int r, c, p, pr, pc, sc, maxcol;
  const Elem *SclTab;
  Vec tmp;

  for (p = sc = 0; p < nrows; p++) {  /* sc is leading col of submatrix */
    pc = xcols;
    /* find pivot */
    for (r = p; r < nrows; r++) {
      for (c = sc; c < pc && !X[r][c]; c++); /* find leading nonzero */
      if (c < pc) {
        pc = c;
```

```
      pr = r;    /* current leader */
      if (c == sc)
        break;   /* found pivot */
    }
  }
  if (pc == xcols)   /* then done; rest is 0 */
    break;
  /* swap rows if nec */
  if (pr > p) {   /* swap rows */
    tmp = X[p];
    X[p] = X[pr];
    X[pr] = tmp;
    if (ycols) {   /* so can reduce single matrix too */
      tmp = Y[p];
      Y[p] = Y[pr];
      Y[pr] = tmp;
    }
  }
  /* normalize: want pivot = 1 (not really nec, but simpler) */
  for (maxcol = xcols; maxcol > pc && !X[p][maxcol - 1]; maxcol--); /* row end */
  if (X[p][pc] != 1) {
    SclTab = ALog + FieldMask - Log[X[p][pc]];
    for (c = pc; c < maxcol; c++)
      X[p][c] = SclTab[Log[X[p][c]]];  /* scale row */
    for (c = 0; c < ycols; c++)
      Y[p][c] = SclTab[Log[Y[p][c]]];  /* scale row */
  }
  /* clear column */
  for (r = p + 1; r < nrows; r++) {
    if (X[r][pc]) {
      SclTab = ALog + Log[X[r][pc]];
      for (c = pc; c < maxcol; c++)
        X[r][c] ^= SclTab[Log[X[p][c]]]; /* subtract pivot row */
      for (c = 0; c < ycols; c++)
```

```
            Y[r][c] ^= SclTab[Log[Y[p][c]]]; /* subtract pivot row */
        }
      }
      /* next */
      sc = pc + 1;
    }
    return p;
}

void
RemoveRows(Mat Rows, Mat X, int rrows, int xrows, int ncols)
{
    /* remove row space of R from matrix X */
    int r, c, p, pc, maxcol, rank;
    const Elem *SclTab;

    rank = ReduceMats(Rows, Rows, rrows, ncols, 0);
    /* find pivot */
    for (p = pc = 0; p < rank; p++) {  /* each pivot row */
      while (pc < ncols && !Rows[p][pc])
        pc++;     /* find pivot */
      for (maxcol = ncols; maxcol > pc && !Rows[p][maxcol - 1]; maxcol--); /* row end */
      /* clear column */
      for (r = 0; r < xrows; r++) {
        if (X[r][pc]) {
          SclTab = ALog + Log[X[r][pc]];
          for (c = pc; c < maxcol; c++)
            X[r][c] ^= SclTab[Log[Rows[p][c]]]; /* subtract pivot row */
        }
      }
    }
}

int
```

```
ReduceEqs(Eq * E, int neqs, int *eqidx)
{
  /* finds U, returns dimension of linkage among specified eqs */
  int i, j, r, c, ntot, rank;
  Mat M, U;

  for (i = ntot = 0; i < neqs; i++)
    ntot += E[eqidx[i]].nrows;

  /* allocate Matrices M, U */
  M = (Mat) malloc(ntot * sizeof(Vec));
  for (i = 0; i < ntot; i++) {
    M[i] = (Vec) malloc(Nvar * sizeof(Elem));
  }
  U = (Mat) malloc(ntot * sizeof(Vec));
  for (i = 0; i < ntot; i++) {
    U[i] = (Vec) calloc(ntot, sizeof(Elem));
    U[i][i] = 1;    /* U is identity matrix */
  }

  /* copy lin eqs into M */
  for (i = r = 0; i < neqs; i++)
    for (j = 0; j < E[eqidx[i]].nrows; j++, r++)
      for (c = 0; c < Nvar; c++)
        M[r][c] = E[eqidx[i]].A[j][c];
#ifdef SHOWMATRICES
  printf("␣ReduceEqs␣input:\n");
  WriteMatrix(M, ntot, Nvar, "A");
#endif
  rank = ReduceMats(M, U, ntot, Nvar, ntot);
  /*#ifdef SHOWMATRICES */
  if (rank < ntot) {
    /* printf(" ReduceEqs output dependencies:\n"); */
// WriteMatrix(U, ntot, ntot, "L");
```

```
    WriteMatrix(U + rank, ntot − rank, ntot, "U");
  }
  /*#endif */

  /* de−allocate Matrices M, U */
  for (i = 0; i < ntot; i++)
    free(U[i]);
  free(U);
  for (i = 0; i < ntot; i++)
    free(M[i]);
  free(M);

  return ntot − rank;
}

int
CompareInts(const void *x, const void *y)
{
  /* for sorting or searching array of ints */
  int a, b;
  a = *((int *) x);
  b = *((int *) y);
  return (a < b) ? −1 : (a > b) ? 1 : 0;
}

int
CompareURows(const void *x, const void *y)
{
  /* for sorting or searching matrix of rows of ints */
  /* NOTE: calling routine MUST set global var ARRAYLENGTH */
  Vec a, b;
  int i;
  a = *((Vec *) x);
  b = *((Vec *) y);
```

```
  for (i = 0; i < ARRAYLENGTH && a[i] == b[i]; i++);
  if (i == ARRAYLENGTH)
    return 0;
  if (a[i] == 0)
    return 1;
  if (b[i] == 0)
    return -1;
  if (a[i] < b[i])
    return -1;
  return 1;
}

int
LinkEqs(Sys * S, int neqs, int *eqidx)
{
  /* finds U, removes known links, creates new links, returns dimension of new linkage
   * among specified eqs */
  int i, j, k, l, kl, r, c, cl, rc, ntot, rank, dim;
  int leqs, lcols, ldim;
  Mat M, U, D, Rows, newU;
  int *eqcol, *list, *includedlinks, nincl, dimincl;
  int **newlist;
  Status stat = OK;

  qsort((void *) eqidx, (size_t) neqs, sizeof(int), CompareInts); /* sort indices */

  /* which cols of U go with which eqs? */
  eqcol = (int *) malloc((neqs + 1) * sizeof(int)); /* alloc eqcol */
  for (i = ntot = 0; i < neqs; i++) {
    eqcol[i] = ntot;
    ntot += S->E[eqidx[i]].nrows;
  }
  eqcol[i] = ntot;    /* end, for later convenience */
#ifdef DEBUG
```

```
    fprintf(stderr, "eqidx:");
    for (i = 0; i < neqs; i++)
      fprintf(stderr, " %d", eqidx[i]);
    fprintf(stderr, "\nmade eq col list:", r);
    for (i = 0; i <= neqs; i++)
      fprintf(stderr, " %d", eqcol[i]);
    fprintf(stderr, " (for %d eqs)\n", neqs);
#endif

    /* allocate Matrices M, U */
    M = (Mat) malloc(ntot * sizeof(Vec));
    for (i = 0; i < ntot; i++)
      M[i] = (Vec) malloc(Nvar * sizeof(Elem));
    U = (Mat) malloc(ntot * sizeof(Vec));
    for (i = 0; i < ntot; i++) {
      U[i] = (Vec) calloc(ntot, sizeof(Elem));
      U[i][i] = 1;    /* U is identity matrix */
    }

    /* copy lin eqs into M */
    for (i = r = 0; i < neqs; i++)
      for (j = 0; j < S->E[eqidx[i]].nrows; j++, r++)
        for (c = 0; c < Nvar; c++)
          M[r][c] = S->E[eqidx[i]].A[j][c];

#ifdef SHOWMATRICES
    printf(" LinkEqs input:\n");
    WriteMatrix(M, ntot, Nvar, "M");
#endif
    rank = ReduceMats(M, U, ntot, Nvar, ntot);
    dim = ntot - rank;    /* dimension of dependency */

    if (dim) {
      /* row reduce dependencies (could sort instead) */
```

```
    D = U + rank;
#ifdef DEBUG
    WriteMatrix(D, dim, ntot, "D");
#endif
#if 0     /* do not sort */
    ReduceMats(D, D, dim, ntot, 0);
    ARRAYLENGTH = ntot;
    qsort((void *) D, (size_t) dim, sizeof(Vec), CompareURows); /* sort rows */
#endif
    /* find current links included */
    includedlinks = (int *) malloc(S->nlinks * sizeof(int)); /* alloc includedlinks */
    nincl = dimincl = 0;
    for (i = 0; i < neqs - 1; i++) {  /* each eq in U might have link */
      for (j = 0; j < S->E[eqidx[i]].nlinks; j++) { /* each link of eq */
        l = S->E[eqidx[i]].linklist[j];
        list = S->L[l].eqlist;
        if (S->L[l].dim <= dim - dimincl /* small enough */
            && list[0] == eqidx[i]  /* not yet checked */
            &&list[S->L[l].neqs - 1] <= eqidx[neqs - 1]) /* not outside range */
          for (k = i + 1, kl = 1; k < neqs; k++){/* scan eqlist to match */
            if (list[kl] == eqidx[k]) {  /* matched next one */
              if (++kl == S->L[l].neqs) { /* got em all */
                includedlinks[nincl++] = l;
                dimincl += S->L[l].dim;
                if (dimincl >= dim) {
                  i = neqs;
                  j = S->nlinks;
                }   /* nothing left */
                break;
              }
            } else if (list[kl] < eqidx[k]) /* skipped one */
                break;
          }
      }
```

```
    }
    /* eliminate included links */
#ifdef DEBUG
    fprintf(stderr, "found␣%d␣included␣links␣of␣total␣dimension␣%d\n", nincl, dimincl);
#endif
    if (nincl && dim > dimincl) {  /* dimension reduced, still positive */
      /* copy included links' dependencies into Rows */
      Rows = (Mat) malloc(dimincl * sizeof(Vec));
      for (i = 0; i < dimincl; i++)
        Rows[i] = (Vec) calloc(ntot, sizeof(Elem));
      for (i = rc = 0; i < nincl; i++) { /* each incl link */
        l = includedlinks[i];
        list = S->L[l].eqlist;
        for (kl = k = cl = 0; kl < S->L[l].neqs; kl++) { /* each eq in link */
          while (eqidx[k] < list[kl])
            k++;    /* find eq */
          c = eqcol[k];
          for (j = 0; j < S->E[eqidx[k]].nrows; j++) /* each col in block */
            for (r = 0; r < S->L[l].dim; r++) /* each row of link */
              Rows[rc + r][c + j] = S->L[l].U[r][cl + j]; /* copy block */
          cl += j;    /* col in link for next eq */
        }
        rc += r;    /* row in copy for next link */
      }
      /* remove included links */
#ifdef DEBUG
      WriteMatrix(Rows, dimincl, ntot, "Rows");
#endif
      RemoveRows(Rows, D, dimincl, dim, ntot);
      for (i = dimincl; i > 0;)
        free(Rows[--i]);
      free(Rows);
      r = ReduceMats(D, D, dim, ntot, 0);
      if (dim != r + dimincl) {
```

```
        fprintf(stderr, "␣ERROR␣in␣LinkEqs;␣reduction␣didn't␣work:␣%d␣+␣%d␣!=␣%
            d,␣links␣may␣be␣incorrect!\n",
                r, dimincl, dim);
        stat = Fail;
      }
    }
    free(includedlinks);
    /* at this point D has dimension dim−dimincl and is reduced */
    dim −= dimincl;
#ifdef SHOWMATRICES
    if (rank < ntot) {
      printf("␣LinkEqs␣output␣dependencies:\n");
      WriteMatrix(D, dim, ntot, "D");
    }
#endif
    if (dim) {    /* then new link(s), check for link dim
                                                    * > 1 */
      /* make new list of eqs for each row; extra for number of eqs, cols */
      newlist = (int **) malloc(dim * sizeof(int *));
      for (i = 0; i < dim; i++)
        newlist[i] = (int *) malloc((neqs + 2) * sizeof(int));
#ifdef DEBUG
      fprintf(stderr, "now␣dim␣=␣%d,␣make␣eq␣lists\n", dim);
#endif
      for (r = 0; r < dim; r++) {  /* each row */
        /* new list of eqs in this row */
        for (c = i = k = lcols = 0; c < ntot;) {
          while (c < ntot && !D[r][c])
            c++;    /* next nonzero */
          if (c < ntot) {   /* found another */
            while (eqcol[k] <= c)
              k++;    /* til beyond */
            newlist[r][i++] = k − 1;  /* save index */
            c = eqcol[k];   /* continue from next */
```

```
            lcols += c − eqcol[k − 1];
          }
        }
        newlist[r][neqs] = i;  /* new neqs */
        newlist[r][neqs + 1] = lcols;  /* new ncols */
      }
#ifdef DEBUG
      fprintf(stderr, "made␣%d␣eq␣lists\n", r);
      for (r = 0; r < dim; r++) {
        fprintf(stderr, "␣list␣%d:", r);
        for (i = 0; i < newlist[r][neqs]; i++)
          fprintf(stderr, "␣%d", newlist[r][i]);
        fprintf(stderr, ",␣n␣=␣%d,␣d␣=␣%d\n", newlist[r][neqs], newlist[r][neqs + 1]);
      }
#endif
      /* make new Link */
      for (r = 0; r < dim;) {  /* each row that starts link */
        leqs = newlist[r][neqs];
        lcols = newlist[r][neqs + 1];
        for (i = r + 1; i < dim  /* find identical lists, assume row−ech */
            && newlist[i][0] == newlist[r][0]
            && newlist[i][neqs] == leqs
            && newlist[i][neqs + 1] == lcols;) {
#ifndef ONEDIMLINKS
          for (j = 1; j < leqs && newlist[i][j] == newlist[r][j]; j++);
          if (j == leqs)
            i++;  /* found twin list, keep looking */
          else
#endif
            break;
        }
        ldim = i − r;  /* now i−r is dimension of new link */
#ifdef DEBUG
```

```
            fprintf(stderr, "about␣to␣make␣new␣link:␣neqs␣=␣%d,␣dim␣=␣%d,␣cols␣=␣%d\
                n", leqs, ldim, lcols);
#endif
            /* allocate & initialize eqlist & U for this link */
            list = (int *) malloc(leqs * sizeof(int));
            newU = (Mat) malloc(dim * sizeof(Vec)); /* new U for link */
            for (j = 0; j < ldim; j++)  /* allocate rows of new U */
              newU[j] = (Vec) malloc(lcols * sizeof(Elem));
            for (i = cl = 0; i < leqs; i++) { /* each eq in link */
              k = newlist[r][i];
              list[i] = eqidx[k];
              for (c = eqcol[k]; c < eqcol[k + 1]; c++, cl++) /* each col in block */
                for (j = 0; j < ldim; j++)  /* each row of link */
                  newU[j][cl] = D[r + j][c]; /* copy block */
            }
            if (NewLink(S, leqs, ldim, lcols, list, newU) != OK)
              return -1;    /* failure signal */
            l = S->nlinks - 1;
#ifdef DEBUG
            fprintf(stderr, "made␣new␣link␣#%d\n", l);
#endif
            r += ldim;
          }
          for (i = dim; i > 0;)
            free(newlist[--i]);
          free(newlist);
        }
      }
      /* de-allocate Matrices M, U */
      for (i = ntot; i > 0;)
        free(U[--i]);
      free(U);
      for (i = ntot; i > 0;)
        free(M[--i]);
```

```
    free(M);
    free(eqcol);

    return dim;
}


/****************** RHS Tools ******************/

void
WriteZ(Link * link)
{
    /* writes Z matrix of link */
    int i, j, k, n;
    Eq *eq;

    printf("␣link␣#");
    n = (int) (link - S->L);   /* link number? */
        if (n < 0 || n >= S->nlinks) {
                printf("␣%d␣??", n); return;}
    else
        printf("%3d", n);
    printf(";␣%d␣eqs:", link->neqs);
    for (i = 0; i < link->neqs; i++)
        printf("␣%d", link->eqlist[i]);
    printf(";␣Z:");

    for (i = 0; i < link->neqs; i++) {
        k = link->eqlist[i];
        printf("\n%3d:␣", k);
        eq = S->E + k;
        for (j = 0; j < link->dim; j++) {
            if (j)
                printf("\n␣␣␣␣␣␣");
```

```
        for (k = 0; k < eq->nrhs; k++)
          printf(OutFormat, link->Z[i][j][k]);
        printf("␣(%d)", (int)(link->Z[i][j][k]) + 1);
      }
    }
    printf("\n");
}

/* rhs_disagree takes a Link and list of indices to RHS and
  returns 0 if agree, else nonzero */
int
rhs_disagree(Link * link, int *irhs)
{
    int len, ieq, row, col, *ires, i, rv;
    Vec res;

    len = intlen(link->dim);
    res = (Vec) calloc(len, sizeof(int));  /* to sum result */
    for (ieq = col = 0; ieq < link->neqs; ieq++)
      for (row = 0; row < link->dim; row++)
        res[row] ^= link->Z[ieq][row][irhs[ieq]];
    ires = (int *) res;
    for (i = 1; i < len; i++)
      ires[0] |= ires[i];    /* combine all nonzero bits */
    rv = ires[0];
    free(res);
    return rv;
}

/* find_rhs takes a Link, an Eq number, and list of indices to RHS and an array for output
  returns number of rhs in given eq that agree with specified RHS values, puts indices in array
      */
int
find_rhs(Link * link, int zeq, int *irhs, int *out)
```

```
{
  int len, ieq, row, zi = −1, i, rv;
  Vec res;
  Eq ∗eq;

  len = intlen(link−>dim);
  res = (Vec) calloc(len, sizeof(int));  /∗ to sum result ∗/
  for (ieq = 0; ieq < link−>neqs; ieq++) {
    if (link−>eqlist[ieq] == zeq) {
      zi = ieq;
      irhs−−;
    } else if (ieq == link−>neqs − 1 && zi == −1) { /∗ didn't find it! (no overrun) ∗/
      fprintf(stderr, "␣find_rhs␣could␣not␣find␣eq␣%d␣in␣link␣%d!\n", zeq, (int) (link −
          S−>L));
      free(res);
      return −1;   /∗ flag error ∗/
    } else
      for (row = 0; row < link−>dim; row++)
        res[row] ^= link−>Z[ieq][row][irhs[ieq]];
  }
  /∗ at this point, res has sum excluding specified Eq. Check which z values match (sum=0) ∗/
  eq = S−>E + zeq;
  for (rv = i = 0; i < eq−>nrhs; i++) {
    for (row = 0; row < link−>dim; row++)
      if (link−>Z[zi][row][i] != res[row])
        break;    /∗ not consistent ∗/
    if (row == link−>dim)
      out[rv++] = i;
  }
  free(res);
  return rv;
}

#include "newagree.h"
```

APPENDIX E:
Square MRHS Equation Creation Code

This appendix contains the C code for the construction of MRHS equations for Small scale variants of the SQUARE algorithm. It consists of a main file (sqr_eqs_encr_f) and a header file (sqr_eqs_io_f).

sqr_eqs_encr_f.c

```
/*
        sqr_eqs.c
        version: 2012 Jun 14
Generate MRHS Equations for
Small Scale Variants of the SQUARE algorithm
Also does the encryption!
Output in field elements, not bits!
Notes: always keysize = block size
optional command line arguments:
        variant (string) = "nrce" to specify small-scale variant of AES:
                        n (hex) is # rounds ( 1 - A; default=A=10 )
                        r (int) is # rows ( 1, 2, 4; default=4 )
                        c (int) is # cols ( 1, 2, 4; default=4 )
                        e (int) is # bits in word ( 2, 4, 8; default=8 )
                        defaults are "8448" for standard SQUARE(8; 4; 4; 8)
        input (hex) = plaintext block (default is zero block)
        output (hex) = key block (default is zero block)
        outfile (string) = filename of output file (default is stdout)
while all the above are optional, you must have one to have the next...

save all the X state data (output of S-box after Transpose) and K key data,
print it out after the equations.
encryption re-organized the to give the X state:
        put Transpose before S-box
```

```
        do NOT do "in place"; rather, put result in new place.
(M') (AMTS)*n (A) rather than
(M'A) (MSTA)*n [where T is Transpose, S is Sbox, M is Mix...]
Does actual KeySchedule and Encrypt.
(Note: keep M' = InvMix rather than modified 1st round.)
*/

#include <stdio.h>
#include <string.h>

//#define OUTPUTBITS // to print output in bits (otherwise in field elements)

#define MAXROUNDS 8
#define MAXROWS 4
#define MAXCOLS 4
#define MAXBITS 8
#define MAXBLOCK MAXROWS*MAXCOLS
#define MAXKEY MAXBLOCK
#define MAXVARS MAXROUNDS*MAXROWS*(MAXCOLS+1)

unsigned char RoundKeys[(MAXROUNDS + 1) * MAXBLOCK];
unsigned char States[(MAXROUNDS + 3) * MAXBLOCK];
unsigned int *Log;
unsigned char *ALog, *Sbox, *Mix, *InvMix, fieldmask;
unsigned char *MixC, *InvMixC, *MixR, *InvMixR;
int nRounds = 8, nRows = 4, nCols = 4, nBits = 8, field, block,
    KeyBits, KeyCols, nKeyCols;
int nEqs, nVars, nKeyVars;
unsigned char PT[MAXBLOCK], CT[MAXBLOCK], Eq[2][MAXVARS], Data[2];
enum InOut { In, Out };
enum VarType { Key, X, State };
int Len, Wid; // for eqs

unsigned int Log8[256] = {
```

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

unsigned char ALog8[256] = {
0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0xF5,0x1F,0x3E,0x7C,0xF8,0x05,0x0A,0x14,
0x28,0x50,0xA0,0xB5,0x9F,0xCB,0x63,0xC6,0x79,0xF2,0x11,0x22,0x44,0x88,0xE5,0x3F,
0x7E,0xFC,0x0D,0x1A,0x34,0x68,0xD0,0x55,0xAA,0xA1,0xB7,0x9B,0xC3,0x73,0xE6,0
    x39,
0x72,0xE4,0x3D,0x7A,0xF4,0x1D,0x3A,0x74,0xE8,0x25,0x4A,0x94,0xDD,0x4F,0x9E,0xC9
    ,
0x67,0xCE,0x69,0xD2,0x51,0xA2,0xB1,0x97,0xDB,0x43,0x86,0xF9,0x07,0x0E,0x1C,0x38,
0x70,0xE0,0x35,0x6A,0xD4,0x5D,0xBA,0x81,0xF7,0x1B,0x36,0x6C,0xD8,0x45,0x8A,0xE1
    ,
```

```
0x37,0x6E,0xDC,0x4D,0x9A,0xC1,0x77,0xEE,0x29,0x52,0xA4,0xBD,0x8F,0xEB,0x23,0
    x46,
0x8C,0xED,0x2F,0x5E,0xBC,0x8D,0xEF,0x2B,0x56,0xAC,0xAD,0xAF,0xAB,0xA3,0xB3,0
    x93,
0xD3,0x53,0xA6,0xB9,0x87,0xFB,0x03,0x06,0x0C,0x18,0x30,0x60,0xC0,0x75,0xEA,0x21,
0x42,0x84,0xFD,0x0F,0x1E,0x3C,0x78,0xF0,0x15,0x2A,0x54,0xA8,0xA5,0xBF,0x8B,0xE3,
0x33,0x66,0xCC,0x6D,0xDA,0x41,0x82,0xF1,0x17,0x2E,0x5C,0xB8,0x85,0xFF,0x0B,0x16,
0x2C,0x58,0xB0,0x95,0xDF,0x4B,0x96,0xD9,0x47,0x8E,0xE9,0x27,0x4E,0x9C,0xCD,0x6F
    ,
0xDE,0x49,0x92,0xD1,0x57,0xAE,0xA9,0xA7,0xBB,0x83,0xF3,0x13,0x26,0x4C,0x98,0xC5
    ,
0x7F,0xFE,0x09,0x12,0x24,0x48,0x90,0xD5,0x5F,0xBE,0x89,0xE7,0x3B,0x76,0xEC,0x2D,
0x5A,0xB4,0x9D,0xCF,0x6B,0xD6,0x59,0xB2,0x91,0xD7,0x5B,0xB6,0x99,0xC7,0x7B,0
    xF6,
0x19,0x32,0x64,0xC8,0x65,0xCA,0x61,0xC2,0x71,0xE2,0x31,0x62,0xC4,0x7D,0xFA,0x01,
};

unsigned char Sbox8[256] = {
0xB1,0xCE,0xC3,0x95,0x5A,0xAD,0xE7,0x02,0x4D,0x44,0xFB,0x91,0x0C,0x87,0xA1,0
    x50,
0xCB,0x67,0x54,0xDD,0x46,0x8F,0xE1,0x4E,0xF0,0xFD,0xFC,0xEB,0xF9,0xC4,0x1A,0
    x6E,
0x5E,0xF5,0xCC,0x8D,0x1C,0x56,0x43,0xFE,0x07,0x61,0xF8,0x75,0x59,0xFF,0x03,0x22,
0x8A,0xD1,0x13,0xEE,0x88,0x00,0x0E,0x34,0x15,0x80,0x94,0xE3,0xED,0xB5,0x53,0x23,
0x4B,0x47,0x17,0xA7,0x90,0x35,0xAB,0xD8,0xB8,0xDF,0x4F,0x57,0x9A,0x92,0xDB,0
    x1B,
0x3C,0xC8,0x99,0x04,0x8E,0xE0,0xD7,0x7D,0x85,0xBB,0x40,0x2C,0x3A,0x45,0xF1,0x42,
0x65,0x20,0x41,0x18,0x72,0x25,0x93,0x70,0x36,0x05,0xF2,0x0B,0xA3,0x79,0xEC,0x08,
0x27,0x31,0x32,0xB6,0x7C,0xB0,0x0A,0x73,0x5B,0x7B,0xB7,0x81,0xD2,0x0D,0x6A,0x26,
0x9E,0x58,0x9C,0x83,0x74,0xB3,0xAC,0x30,0x7A,0x69,0x77,0x0F,0xAE,0x21,0xDE,0xD0
    ,
0x2E,0x97,0x10,0xA4,0x98,0xA8,0xD4,0x68,0x2D,0x62,0x29,0x6D,0x16,0x49,0x76,0xC7,
0xE8,0xC1,0x96,0x37,0xE5,0xCA,0xF4,0xE9,0x63,0x12,0xC2,0xA6,0x14,0xBC,0xD3,0x28,
```

```
0xAF,0x2F,0xE6,0x24,0x52,0xC6,0xA0,0x09,0xBD,0x8C,0xCF,0x5D,0x11,0x5F,0x01,0xC5
    ,
0x9F,0x3D,0xA2,0x9B,0xC9,0x3B,0xBE,0x51,0x19,0x1F,0x3F,0x5C,0xB2,0xEF,0x4A,0
    xCD,
0xBF,0xBA,0x6F,0x64,0xD9,0xF3,0x3E,0xB4,0xAA,0xDC,0xD5,0x06,0xC0,0x7E,0xF6,0
    x66,
0x6C,0x84,0x71,0x38,0xB9,0x1D,0x7F,0x9D,0x48,0x8B,0x2A,0xDA,0xA5,0x33,0x82,0x39,
0xD6,0x78,0x86,0xFA,0xE4,0x2B,0xA9,0x1E,0x89,0x60,0x6B,0xEA,0x55,0x4C,0xF7,0xE2,
};

unsigned char invSbox8[256] = {
0x35,0xBE,0x07,0x2E,0x53,0x69,0xDB,0x28,0x6F,0xB7,0x76,0x6B,0x0C,0x7D,0x36,0x8B,
0x92,0xBC,0xA9,0x32,0xAC,0x38,0x9C,0x42,0x63,0xC8,0x1E,0x4F,0x24,0xE5,0xF7,0xC9,
0x61,0x8D,0x2F,0x3F,0xB3,0x65,0x7F,0x70,0xAF,0x9A,0xEA,0xF5,0x5B,0x98,0x90,0xB1,
0x87,0x71,0x72,0xED,0x37,0x45,0x68,0xA3,0xE3,0xEF,0x5C,0xC5,0x50,0xC1,0xD6,0xCA
    ,
0x5A,0x62,0x5F,0x26,0x09,0x5D,0x14,0x41,0xE8,0x9D,0xCE,0x40,0xFD,0x08,0x17,0x4A,
0x0F,0xC7,0xB4,0x3E,0x12,0xFC,0x25,0x4B,0x81,0x2C,0x04,0x78,0xCB,0xBB,0x20,0xBD
    ,
0xF9,0x29,0x99,0xA8,0xD3,0x60,0xDF,0x11,0x97,0x89,0x7E,0xFA,0xE0,0x9B,0x1F,0xD2,
0x67,0xE2,0x64,0x77,0x84,0x2B,0x9E,0x8A,0xF1,0x6D,0x88,0x79,0x74,0x57,0xDD,0xE6,
0x39,0x7B,0xEE,0x83,0xE1,0x58,0xF2,0x0D,0x34,0xF8,0x30,0xE9,0xB9,0x23,0x54,0x15,
0x44,0x0B,0x4D,0x66,0x3A,0x03,0xA2,0x91,0x94,0x52,0x4C,0xC3,0x82,0xE7,0x80,0xC0,
0xB6,0x0E,0xC2,0x6C,0x93,0xEC,0xAB,0x43,0x95,0xF6,0xD8,0x46,0x86,0x05,0x8C,0xB0,
0x75,0x00,0xCC,0x85,0xD7,0x3D,0x73,0x7A,0x48,0xE4,0xD1,0x59,0xAD,0xB8,0xC6,0
    xD0,
0xDC,0xA1,0xAA,0x02,0x1D,0xBF,0xB5,0x9F,0x51,0xC4,0xA5,0x10,0x22,0xCF,0x01,0
    xBA,
0x8F,0x31,0x7C,0xAE,0x96,0xDA,0xF0,0x56,0x47,0xD4,0xEB,0x4E,0xD9,0x13,0x8E,0x49
    ,
0x55,0x16,0xFF,0x3B,0xF4,0xA4,0xB2,0x06,0xA0,0xA7,0xFB,0x1B,0x6E,0x3C,0x33,0
    xCD,
0x18,0x5E,0x6A,0xD5,0xA6,0x21,0xDE,0xFE,0x2A,0x1C,0xF3,0x0A,0x1A,0x19,0x27,0
    x2D,
```

```
};

unsigned int Log4[16] = {
 0x0, 0x0, 0x1, 0xC, 0x2, 0x9, 0xD, 0x7, 0x3, 0x4, 0xA, 0x5, 0xE, 0xB, 0x8, 0x6,
};

unsigned char ALog4[16] = {
 0x1, 0x2, 0x4, 0x8, 0x9, 0xB, 0xF, 0x7, 0xE, 0x5, 0xA, 0xD, 0x3, 0x6, 0xC, 0x1,
};

unsigned char Sbox4[16] = {
 0x9, 0x2, 0x5, 0x1, 0xB, 0x8, 0xD, 0x3, 0x4, 0xE, 0xC, 0x7, 0xF, 0xA, 0x0, 0x6,
};

unsigned char invSbox4[16] = {
 0xE, 0x3, 0x1, 0x7, 0x8, 0x2, 0xF, 0xB, 0x5, 0x0, 0xD, 0x4, 0xA, 0x6, 0x9, 0xC,
};

unsigned int Log2[4] = {
    0, 0, 1, 2,
};

unsigned char ALog2[4] = {
    1, 2, 3, 1,
};

unsigned char Sbox2[4] = {
    2, 1, 3, 0,
};

unsigned char invSbox2[4] = {
    3, 1, 0, 2,
};
```

```
unsigned char Mix4[8] = {
0x2,0x3,0x1,0x1,0x2,0x3,0x1,0x1,
};

unsigned char InvMix4[8] = {
0xE,0xB,0xD,0x9,0xE,0xB,0xD,0x9,
};

unsigned char InvMix42[8] = {
0x0,0x2,0x3,0x0,0x0,0x0,0x2,0x3,0x0,
};

unsigned char Mix2[4] = {
0x3,0x2,0x3,0x2,
};

unsigned char Mix1[2] = {
0x1,0x1,
};

// multiply by "2" in field
#define POLY8 0xF5
#define POLY4 0x19
#define POLY2 0x07
unsigned char HIBIT, POLY;
unsigned char mul2 (unsigned char x) {
    unsigned char y;
    y = x << 1;
    if ( x & HIBIT ) y ^= POLY;
    return( y );
}

// multiply two bytes in field
unsigned char mul(unsigned char x, unsigned char y)
```

```
{
    if (x && y)
        return (ALog[(Log[x] + Log[y]) % (field − 1)]);
    else
        return (0);
}

#include "sqr_eqs_io_f.h" // include field I/O package

// set up specific small−scale variant of AES
// assumes main() already set: nRounds, nRows, nCols, nBits
int setup(void)
{
    int returnval = 0;

// check parameters for validity
    if (nRounds < 1 || nRounds > MAXROUNDS) {
        nRounds = MAXROUNDS;
        returnval = 1;
    }
    switch (nBits) {
    case 2:
        Log = Log2;
        ALog = ALog2;
        Sbox = Sbox2;
        POLY = POLY2;
        field = 4;
        break;
    case 4:
        Log = Log4;
        ALog = ALog4;
        Sbox = Sbox4;
        POLY = POLY4;
        field = 16;
```

```
        break;
    default:
        nBits = 8;
        returnval = 1;  // if bad value, use default, fall thru
    case 8:
        Log = Log8;
        ALog = ALog8;
        Sbox = Sbox8;
        POLY = POLY8;
        field = 256;
        break;
    }
    switch (nRows) {
    case 1:
        MixC = InvMixC = Mix1;
        break;
    case 2:
        MixC = InvMixC = Mix2;
        break;
    default:
        nRows = 4;
        returnval = 1;  // if bad value, use default, fall thru
    case 4:
        MixC = Mix4;
        InvMixC = (nBits == 2) ? InvMix42 : InvMix4;
        break;
    }
  switch (nCols) {
    case 1:
        MixR = InvMixR = Mix1;
        break;
    case 2:
        MixR = InvMixR = Mix2;
        break;
```

```
    default:
        nCols = 4;
        returnval = 1;  // if bad value, use default, fall thru
    case 4:
        MixR = Mix4;
        InvMixR = (nBits == 2) ? InvMix42 : InvMix4;
        break;
    }
    MixC += nRows; InvMixC += nRows; // so can slide index backwards
    MixR += nCols; InvMixR += nCols; // so can slide index backwards
    Mix = MixC ; InvMix = InvMixC; // TEMPO
    fieldmask = field − 1;
    HIBIT = 1 << (nBits − 1);
    setScale(); // set up bit matrices for scalars
    block = nRows ∗ nCols;
    KeyBits = block ∗ nBits;
    nKeyCols = (nRounds + 1) ∗ nCols;
    nKeyVars = block; // key sched is linear!
    nVars = nKeyVars + block ∗ (nRounds − 1);
    nEqs = nVars;

    return returnval;
}

#define RC(r) (ALog[r−1])

/∗
 simplify key sched: block by block
 allow rectangles
 ∗/
int KeySchedule(unsigned char Key[])
{
  int returnval = 0;
  int i, r, c, round;
```

```
// unsigned char col[MAXROWS], t;

   KeyCols = nCols;
   /* Copy key */
   for (i = 0; i < block; i++)
      RoundKeys[i] = Key[i];

   for (round = 1; round <= nRounds; round++) {
      /* calculate new columns until enough */
      if (round&1) {Len = nCols; Wid = nRows; Mix = MixR;}
      else {Len = nRows; Wid = nCols; Mix = MixC;}
      for (r = 1; r < Len; r++) // col 0 = rotate prev col
         RoundKeys[round*block+ r-1] = RoundKeys[round*block-Len +r];
      RoundKeys[round*block+ r-1] = RoundKeys[round*block-Len +0];
      RoundKeys[round*block+ 0] ^= RC(round); // + round key
      if (Wid>1) { // need to handle KeyCols = 1 differently
         for (r = 0; r < Len; r++)
            RoundKeys[round*block+ r] ^= RoundKeys[(round-1)*block+ r];
         for (c=1; c<Wid; c++)
            for (r = 0; r < Len; r++)
               RoundKeys[round*block+ c*Len+r] =
               RoundKeys[round*block+ (c-1)*Len+r] ^ RoundKeys[(round-1)*block+ c*Len+
                  r];
      } // more cols
   } // round loop

   return returnval;
}

// do pre-round on block: (M')
void preround(unsigned char State[])
{
      unsigned char t[MAXROWS];
      int i, r, c;
```

```
    for (c = 0; c < nCols; c++) {
        for (r = 0; r < nRows; r++)
            for (t[r] = i = 0; i < nRows; i++)
                t[r] ^= mul( State[c*nRows+i],
                    InvMixC[i-r]); // InvMixColumns
        for (r = 0; r < nRows; r++)
            State[block + c*nRows+r] = t[r];
    }
}

// do one round on block: (AMTS)
void doround(unsigned char State[], unsigned char roundKey[], int round)
{
  unsigned char t[MAXROWS];
  int i, r, c;

  if (round&1) {Len = nCols; Wid = nRows; Mix = MixR;}
  else {Len = nRows; Wid = nCols; Mix = MixC;}
  for (c = 0; c < Wid; c++) {
    for (r = 0; r < Len; r++)
      for (t[r] = i = 0; i < Len; i++)
        t[r] ^= mul( State[c*Len+i] ^ roundKey[c*Len+i], // AddRoundKey
                    Mix[i-r]); // MixColumns
    for (r = 0; r < Len; r++)
      State[block + r*Wid+c] = t[r]; // Transpose
  }

  State += block;
  for (i = 0; i < block; i++)
    State[i] = Sbox[State[i]]; // SubBytes
}

// do post-round on block: (A)
void postround(unsigned char State[], unsigned char roundKey[])
```

```
{
    int i;
    for (i = 0; i < block; i++)
        State[block + i] = State[i] ^ roundKey[i]; // AddRoundKey
}

// encrypt block (NOT in place − keep output of each S−box)
void encrypt( void )
{
    int i, round;

    for (i = 0; i < block; i++)
        States[i] = PT[i];  // copy PT in
    preround(States);
    for (round = 0; round < nRounds; round++) {
        doround( States + (round+1)∗block, RoundKeys + round∗block, round);
    }
    postround(States + (round+1)∗block, RoundKeys + round∗block);
    for (i = 0; i < block; i++)
        CT[i] = States[(round+2)∗block + i]; // copy CT out
}

void NewEq( void )
{
    int i, r;

    for (r = In; r <= Out; r++) {
    Data[r] = 0;
    for (i = 0; i < nVars; i++)
        Eq[r][i] = 0;
    }
}

int VarNum( enum VarType var,
```

```
            int round, int col, int row )
{
  int i;
  i = col*(round&1 ? nCols : nRows)+row;
  switch (var) {
    case Key:
      return ( i );
    case X:
      return ( round*block + i );
  }
  return ( 0 ); // dummy
}

void AddVar( enum InOut line, enum VarType var,
          int round, int col, int row, int scale )
{
  int r, i;
  int Len, Wid; // local copies for local recursive "round"
  unsigned char *Mix;

  if (round&1) {Len = nCols; Wid = nRows; Mix = MixR;}
  else {Len = nRows; Wid = nCols; Mix = MixC;}
  switch (var) {
    case Key:
      i = col * Len + row;
      if (round == 0) // key var
        Eq[line][ VarNum( Key, round, col, row ) ] ^= scale;
      else if (col == 0) { // F(col)
        if (Wid > 1)
          AddVar( line, Key, round-1, i/Wid, i%Wid, scale );
        i = block-Len + (row+1)%Len;
        AddVar( line, Key, round-1, i/Wid, i%Wid, scale );
        if (row == 0)
          Data[line] ^= mul( RC(round), scale ); // track round constants
```

```
        }
        else {
          AddVar( line, Key, round−1, i/Wid, i%Wid, scale );
          AddVar( line, Key, round, col−1, row, scale );
        }
        break;
      case X:
        Eq[line][ VarNum( X, round, col, row ) ] ^= scale;
        break;
      case State: // scale must be 1; Mix (X+K)
        if (round > 0)
          for (r = 0; r < Len; r++)
            AddVar( line, X, round, col, r, Mix[r−row] );
        for (r = 0; r < Len; r++)
          AddVar( line, Key, round, col, r, Mix[r−row] );
        break;
    }
}

// do only round #0 on block
void doonlyroundEqs(int round)
{
  int r, c;

  Len = nRows; Wid = nCols; Mix = MixC;;
  for (r = 0; r < Len; r++) {
    for (c = 0; c < Wid; c++) {
      NewEq();
      AddVar( In, State, round, c, r, 1);
      AddVar( Out, Key, round+1, r, c, 1); // T
      Data[ In ] ^= PT[ c∗Len + r ];
      Data[ Out ] ^= CT[ r∗Wid + c ]; // T
      WriteEq();
    }
```

```
    }
  }

  // do round #0 on block
  void doround0Eqs(int round)
  {
    int r, c;

    Len = nRows; Wid = nCols; Mix = MixC;
    for (r = 0; r < Len; r++) {
      for (c = 0; c < Wid; c++) {
        NewEq();
        AddVar( In, State, round, c, r, 1);
        AddVar( Out, X, round+1, r, c, 1); // T
        Data[ In ] ^= PT[ c*Len + r ];
        WriteEq();
      }
    }
  }

  // do one round on block
  void doroundEqs(int round)
  {
    int r, c;

    if (round&1) {Len = nCols; Wid = nRows; Mix = MixR;}
    else {Len = nRows; Wid = nCols; Mix = MixC;}
    for (r = 0; r < Len; r++) {
      for (c = 0; c < Wid; c++) {
        NewEq();
        AddVar( In, State, round, c, r, 1);
        AddVar( Out, X, round+1, r, c, 1); // T
        WriteEq();
      }
```

```
    }
}

// do round #n on block
void doroundnEqs(int round)
{
    int r, c;

    if (round&1) {Len = nCols; Wid = nRows; Mix = MixR;}
    else {Len = nRows; Wid = nCols; Mix = MixC;}
    for (r = 0; r < Len; r++) {
        for (c = 0; c < Wid; c++) {
            NewEq();
            AddVar( In, State, round, c, r, 1);
            AddVar( Out, Key, round+1, r, c, 1); // T
            Data[ Out ] ^= CT[ r*Wid + c ]; // T
            WriteEq();
        }
    }
}

void EncryptEqs(void)
{
    int round;

    if ( nRounds == 1 ) {
        doonlyroundEqs(0);
        return;
        }
    doround0Eqs(0);
    for (round = 1; round < nRounds-1; round++) {
        doroundEqs(round);
    }
    doroundnEqs(round);
```

```
}

int main(int argc, char *argv[])
{
unsigned char Key[MAXBLOCK];

    if (argc > 1) {
        sscanf(argv[1], "%1x%1d%1d%1d",
                    &nRounds, &nRows, &nCols, &nBits);
    }
    fprintf(stderr, "␣nRounds=%d,␣nRows=%d,␣nCols=%d,␣nBits=%d\n",
            nRounds, nRows, nCols, nBits);
    if (setup())
        fprintf(stderr,
                "Bad␣parameter(s);␣now:\n␣nRounds=%d,␣nRows=%d,␣nCols=%d,␣
                    nBits=%d\n",
                nRounds, nRows, nCols, nBits);
// by default KeyBits = bits in block
        ReadBlock( (argc > 2) ? argv[2] :
                    "000102030405060708090A0B0C0D0E0F", PT);
        ReadBlock( (argc > 3) ? argv[3] :
                    "000102030405060708090A0B0C0D0E0F", Key);
    if (argc > 4)
        if (freopen(argv[4], "w", stdout) != stdout) {
            fprintf(stderr, "Could␣not␣open␣output␣file␣%s\n", argv[4]);
            return 1;
        }

    KeySchedule(Key);
    encrypt();

    WriteSystemHeader();
    EncryptEqs();
```

```
    WriteVars();
    printf("␣nRounds=%d,␣nRows=%d,␣nCols=%d,␣nBits=%d\n",
            nRounds, nRows, nCols, nBits);
    WriteKeys();
    WriteStates( );

    return (0);
}
```

sqr_eqs_io_f.h

```
/*
        sqr_eqs_io.h
        version: 2012 Apr 15
*/

/*
Write Equation, field elements version
*/

char *FieldFormat;
void setScale(void){ // set up FieldFormat
  FieldFormat = (nBits > 4)? "%02X" : "%01X";
}

void writeValue(unsigned int x){
  /* writes field element x */

      printf( FieldFormat, x );
}

void WriteSystemHeader( void ) // note third number on top line: #bits in field
{
    printf("␣␣%d␣␣%d␣␣%d\n", nVars, nEqs, nBits );
}
```

```
void WriteEq( void )
{
    int i, r;

    printf("␣%d␣%d\n", 2, field);
    for (r = In; r <= Out; r++) {
    for (i = 0; i < nVars; i++)
        writeValue( Eq[r][i] );
    printf("\n");
    }
    for ( i=0; i < field; i++ ) {
        if ( !(i&15) && i ) printf("\n");
        writeValue( i ^ Data[In] );
        writeValue( Sbox[i] ^ Data[Out] );
        }
        printf("\n");
}

// read hex data into block
void ReadBlock( char str[], unsigned char T[] )
{
    char *j;
    int i, word;

    switch (nBits) {
    case 2:
    if ( block == 1 ) { // this is the only case with an odd number of words
        word = 0;
        sscanf( str, "%1x", &word );
        T[0] = (unsigned char) (word & fieldmask);
        i = 1; break;
        }
        for (i = 0, j = str; i < block; ) {
```

```
            if ( sscanf( j++, "%1x", &word ) != 1 ) break;
            T[i++] = (unsigned char) ((word>>2) & fieldmask);
            T[i++] = (unsigned char) (word & fieldmask);
        }
        break;
    case 4:
        for (i = 0; i < block; i++) {
            if ( sscanf( str+i, "%1x", &word ) != 1 ) break;
            T[i] = (unsigned char) word;
        }
        break;
    case 8:
        for (i = 0; i < block; i++) {
            if ( sscanf( str+2*i, "%2x", &word ) != 1 ) break;
            T[i] = (unsigned char) word;
        }
        break;
    default:
                i = 0;
    }
        for ( ; i < block; i++)
            T[i] = 0;
}

// write hex data from block
void WriteBlock( unsigned char T[] )
{
    int i;

        for (i = 0; i < block; i++)
            writeValue( T[i] & fieldmask );
}

// write Round Keys
```

```
void WriteKeys( void )
{
    int round;

    printf( "Round␣Keys:\n");
    for (round = 0; round <= nRounds; round++) {
            printf( "␣%2d␣:␣", round);
            WriteBlock( RoundKeys + round∗block );
            printf("\n");
            }
}

// write States
void WriteStates( void )
{
    int round;

    printf( "States:␣\n");
    for (round = 0; round <= nRounds+2; round++) {
            printf( "␣%2d␣:␣", round−1);
            WriteBlock( States + round∗block );
            printf("\n");
            }
}

// write Variables
void WriteVars( void )
{
    int i, round;

    printf( "Variables:\n");
        for (i = 0; i < block; i++)
            writeValue( RoundKeys[i] );
            printf("\n");
```

```
    for (round = 1; round < nRounds; round++) {
        for (i = 0; i < block; i++)
            writeValue( States[(round+1)*block + i] );
            printf("\n");
            }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F:
Square MRHS Algorithm Code

The main code does not change to solve the Square variants using our methods. The program is compiled from the same code see in Appendix D except with the following header file (sqfieldarith) to replace the other field arithmetic definitions (fieldarith).

sqfieldarith.h

```
/*
        sqfieldarith.h
        version: 2013 Jan 08
** SQUARE block cipher version!!
 different representation of fields:
 "h" poly basis in 2^8 : h^8+h^7+h^6+h^5+h^4+h^2+1 = 0
 "beta" poly basis in 2^4 : b^4+b^3+1 = 0
 "Omega" poly basis in 2^2 : w^2+w+1 = 0
*/

/* field multiplication done through lookup logs, add logs, lookup antilog
to avoid doing modulo (q-1) the antilog table is doubled
to avoid testing for zero, log(0) = infinity (really 2q-1)
so need to pad end of antilog table with zeros up to 3q-3, also at 4q-2
here, rounded up size of tables for alignment reasons
Note: below embed smaller antilog tables in unused portion of bigger ones
Warning: in macros below, inversion or division by zero gives wrong answer,
as does zero to a power
*/

#define fmul(x,y) (ALog[ Log[x]+Log[y] ]) /* field x*y */
#define finv( x ) (ALog[ FieldMask-Log[x] ]) /* field 1/x (x != 0) */
#define fdiv(x,y) (ALog[ Log[x]+FieldMask-Log[y] ]) /* field x/y (y != 0) */
#define fpow(x,y) (ALog[ (Log[x]*(y)) % FieldMask +FieldMask]) /* (field x)^(integer y) */
```

```
const Elem *ALog;
const int *Log;

const Elem ALogs[1024] = {
0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0xF5,0x1F,0x3E,0x7C,0xF8,0x05,0x0A,0x14,
0x28,0x50,0xA0,0xB5,0x9F,0xCB,0x63,0xC6,0x79,0xF2,0x11,0x22,0x44,0x88,0xE5,0x3F,
0x7E,0xFC,0x0D,0x1A,0x34,0x68,0xD0,0x55,0xAA,0xA1,0xB7,0x9B,0xC3,0x73,0xE6,0
    x39,
0x72,0xE4,0x3D,0x7A,0xF4,0x1D,0x3A,0x74,0xE8,0x25,0x4A,0x94,0xDD,0x4F,0x9E,0xC9
    ,
0x67,0xCE,0x69,0xD2,0x51,0xA2,0xB1,0x97,0xDB,0x43,0x86,0xF9,0x07,0x0E,0x1C,0x38,
0x70,0xE0,0x35,0x6A,0xD4,0x5D,0xBA,0x81,0xF7,0x1B,0x36,0x6C,0xD8,0x45,0x8A,0xE1
    ,
0x37,0x6E,0xDC,0x4D,0x9A,0xC1,0x77,0xEE,0x29,0x52,0xA4,0xBD,0x8F,0xEB,0x23,0
    x46,
0x8C,0xED,0x2F,0x5E,0xBC,0x8D,0xEF,0x2B,0x56,0xAC,0xAD,0xAF,0xAB,0xA3,0xB3,0
    x93,
0xD3,0x53,0xA6,0xB9,0x87,0xFB,0x03,0x06,0x0C,0x18,0x30,0x60,0xC0,0x75,0xEA,0x21,
0x42,0x84,0xFD,0x0F,0x1E,0x3C,0x78,0xF0,0x15,0x2A,0x54,0xA8,0xA5,0xBF,0x8B,0xE3,
0x33,0x66,0xCC,0x6D,0xDA,0x41,0x82,0xF1,0x17,0x2E,0x5C,0xB8,0x85,0xFF,0x0B,0x16,
0x2C,0x58,0xB0,0x95,0xDF,0x4B,0x96,0xD9,0x47,0x8E,0xE9,0x27,0x4E,0x9C,0xCD,0x6F
    ,
0xDE,0x49,0x92,0xD1,0x57,0xAE,0xA9,0xA7,0xBB,0x83,0xF3,0x13,0x26,0x4C,0x98,0xC5
    ,
0x7F,0xFE,0x09,0x12,0x24,0x48,0x90,0xD5,0x5F,0xBE,0x89,0xE7,0x3B,0x76,0xEC,0x2D,
0x5A,0xB4,0x9D,0xCF,0x6B,0xD6,0x59,0xB2,0x91,0xD7,0x5B,0xB6,0x99,0xC7,0x7B,0
    xF6,
0x19,0x32,0x64,0xC8,0x65,0xCA,0x61,0xC2,0x71,0xE2,0x31,0x62,0xC4,0x7D,0xFA,
0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0xF5,0x1F,0x3E,0x7C,0xF8,0x05,0x0A,0x14,
0x28,0x50,0xA0,0xB5,0x9F,0xCB,0x63,0xC6,0x79,0xF2,0x11,0x22,0x44,0x88,0xE5,0x3F,
0x7E,0xFC,0x0D,0x1A,0x34,0x68,0xD0,0x55,0xAA,0xA1,0xB7,0x9B,0xC3,0x73,0xE6,0
    x39,
0x72,0xE4,0x3D,0x7A,0xF4,0x1D,0x3A,0x74,0xE8,0x25,0x4A,0x94,0xDD,0x4F,0x9E,0xC9
    ,
```

```
0x67,0xCE,0x69,0xD2,0x51,0xA2,0xB1,0x97,0xDB,0x43,0x86,0xF9,0x07,0x0E,0x1C,0x38,
0x70,0xE0,0x35,0x6A,0xD4,0x5D,0xBA,0x81,0xF7,0x1B,0x36,0x6C,0xD8,0x45,0x8A,0xE1
    ,
0x37,0x6E,0xDC,0x4D,0x9A,0xC1,0x77,0xEE,0x29,0x52,0xA4,0xBD,0x8F,0xEB,0x23,0
    x46,
0x8C,0xED,0x2F,0x5E,0xBC,0x8D,0xEF,0x2B,0x56,0xAC,0xAD,0xAF,0xAB,0xA3,0xB3,0
    x93,
0xD3,0x53,0xA6,0xB9,0x87,0xFB,0x03,0x06,0x0C,0x18,0x30,0x60,0xC0,0x75,0xEA,0x21,
0x42,0x84,0xFD,0x0F,0x1E,0x3C,0x78,0xF0,0x15,0x2A,0x54,0xA8,0xA5,0xBF,0x8B,0xE3,
0x33,0x66,0xCC,0x6D,0xDA,0x41,0x82,0xF1,0x17,0x2E,0x5C,0xB8,0x85,0xFF,0x0B,0x16,
0x2C,0x58,0xB0,0x95,0xDF,0x4B,0x96,0xD9,0x47,0x8E,0xE9,0x27,0x4E,0x9C,0xCD,0x6F
    ,
0xDE,0x49,0x92,0xD1,0x57,0xAE,0xA9,0xA7,0xBB,0x83,0xF3,0x13,0x26,0x4C,0x98,0xC5
    ,
0x7F,0xFE,0x09,0x12,0x24,0x48,0x90,0xD5,0x5F,0xBE,0x89,0xE7,0x3B,0x76,0xEC,0x2D,
0x5A,0xB4,0x9D,0xCF,0x6B,0xD6,0x59,0xB2,0x91,0xD7,0x5B,0xB6,0x99,0xC7,0x7B,0
    xF6,
0x19,0x32,0x64,0xC8,0x65,0xCA,0x61,0xC2,0x71,0xE2,0x31,0x62,0xC4,0x7D,0xFA, 0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /* ALog4 = ALogs+3*256 */
 0x1, 0x2, 0x4, 0x8, 0x9, 0xB, 0xF, 0x7, 0xE, 0x5, 0xA, 0xD, 0x3, 0x6, 0xC,
 0x1, 0x2, 0x4, 0x8, 0x9, 0xB, 0xF, 0x7, 0xE, 0x5, 0xA, 0xD, 0x3, 0x6, 0xC, 0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /* ALog2 = ALogs+3*256+3*16 */
   1, 2, 3,
   1, 2, 3, 0,0,
0,0,0,0, /* ALog1 = ALogs+3*256+3*16+3*4 */
   1,
   1, 0,0,
```

```
/*0,0,0,0, */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
};

const int Log8[256] = {
 511,0x00,0x01,0x86,0x02,0x0D,0x87,0x4C,0x03,0xD2,0x0E,0xAE,0x88,0x22,0x4D,0x93,
0x04,0x1A,0xD3,0xCB,0x0F,0x98,0xAF,0xA8,0x89,0xF0,0x23,0x59,0x4E,0x35,0x94,0x09,
0x05,0x8F,0x1B,0x6E,0xD4,0x39,0xCC,0xBB,0x10,0x68,0x99,0x77,0xB0,0xDF,0xA9,0x72,
0x8A,0xFA,0xF1,0xA0,0x24,0x52,0x5A,0x60,0x4F,0x2F,0x36,0xDC,0x95,0x32,0x0A,0x1F,
0x06,0xA5,0x90,0x49,0x1C,0x5D,0x6F,0xB8,0xD5,0xC1,0x3A,0xB5,0xCD,0x63,0xBC,0
    x3D,
0x11,0x44,0x69,0x81,0x9A,0x27,0x78,0xC4,0xB1,0xE6,0xE0,0xEA,0xAA,0x55,0x73,0xD8,
0x8B,0xF6,0xFB,0x16,0xF2,0xF4,0xA1,0x40,0x25,0x42,0x53,0xE4,0x5B,0xA3,0x61,0xBF,
0x50,0xF8,0x30,0x2D,0x37,0x8D,0xDD,0x66,0x96,0x18,0x33,0xEE,0x0B,0xFD,0x20,0xD0,
0x07,0x57,0xA6,0xC9,0x91,0xAC,0x4A,0x84,0x1D,0xDA,0x5E,0x9E,0x70,0x75,0xB9,0x6C
    ,
0xD6,0xE8,0xC2,0x7F,0x3B,0xB3,0xB6,0x47,0xCE,0xEC,0x64,0x2B,0xBD,0xE2,0x3E,0
    x14,
0x12,0x29,0x45,0x7D,0x6A,0x9C,0x82,0xC7,0x9B,0xC6,0x28,0x7C,0x79,0x7A,0xC5,0x7B,
0xB2,0x46,0xE7,0x7E,0xE1,0x13,0xEB,0x2A,0xAB,0x83,0x56,0xC8,0x74,0x6B,0xD9,0x9D
    ,
0x8C,0x65,0xF7,0x2C,0xFC,0xCF,0x17,0xED,0xF3,0x3F,0xF5,0x15,0xA2,0xBE,0x41,0xE3
    ,
0x26,0xC3,0x43,0x80,0x54,0xD7,0xE5,0xE9,0x5C,0xB7,0xA4,0x48,0x62,0x3C,0xC0,0xB4,
0x51,0x5F,0xF9,0x9F,0x31,0x1E,0x2E,0xDB,0x38,0xBA,0x8E,0x6D,0xDE,0x71,0x67,0x76,
0x97,0xA7,0x19,0xCA,0x34,0x08,0xEF,0x58,0x0C,0x4B,0xFE,0x85,0x21,0x92,0xD1,0xAD
    ,
};
```

```
const int Log4[16] = {
   31, 0x0, 0x1, 0xC, 0x2, 0x9, 0xD, 0x7, 0x3, 0x4, 0xA, 0x5, 0xE, 0xB, 0x8, 0x6,
};


const int Log2[4] = {
    7, 0, 1, 2,
};

const int Log1[2] = {
    3, 0,
};
```

THIS PAGE INTENTIONALLY LEFT BLANK

REFERENCES

[1] *Advanced Encryption Standard (AES)*, NIST Std. FIPS 197, 2001.

[2] A. Kak. (January 31, 2013) "Lecture 8: AES: The advanced encryption standard". Lecture for class, Computer Science Department, Purdue University at West Lafayette, IN. [Online]. Available: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf.

[3] R. Smith, "Deciphering the advanced encryption standard," *Network Magazine*, vol. 16, pp. 96–101, 2001.

[4] A. Kerckhoff, "La cryptographie militaire," *Journel des Sciences Militaires*, vol. IX, pp. 5–38, 1883.

[5] M. Kreuzer, "Algebraic attacks galore!" *Groups Complexity Cryptology*, vol. 1, no. 2, pp. 231–259, 2009.

[6] Encyclopedia Britannica, s.v. "Claude Shannon". [Online]. Available: http://www.britannica.com/EBchecked/topic/538577/Claude-Shannon.

[7] C. E. Shannon, "Communication theory of secrecy systems," *Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[8] H. Raddum and I. Semaev, "Solving MRHS linear equations," in *Proc. of WCC*, 2007, pp. 323–332.

[9] J. B. Fraleigh, *A First Course in Abstract Algebra*, 7th ed. New York City, NY: Addison-Wesley, 2002.

[10] G. Chartrand and P. Zhang, *Introduction to Graph Theory*. Boston, MA: McGraw Hill Higher Education, 2005, ch. 1-7.

[11] B. Schneier and P. Sutherland, *Applied Cryptography: Protocols, Algorithms, and Source code in C*. Somerset, NJ: John Wiley & Sons, 1995.

[12] RSA official website. [Online]. Available: http://www.rsa.com/rsalabs/

[13] Department of Commerce, National Institute of Standards and Technology. (1997, Jan) Announcing development of federal information processing standard for advanced encryption standard. [Online]. Available: http://csrc.nist.gov/archive/aes/pre-round1/aes_9701.txt

[14] J. Nechvatal, E. Barker, D. Dodson, M. Dworkin, J. Foti, E. Roback *et al.*, "Status report on the first round of the development of the advanced encryption standard," *Journal Of Research of the National Institute of Standards and Technology*, vol. 104, no. 5, pp. 435–460, 1999.

[15] C. Sybrandy, J. MacDonald *et al.*, "Public comments regarding the advanced encryption standard (AES) development effort round 2." [Online]. Available: http://csrc.nist.gov/archive/aes/index.html

[16] Public comments regarding the advanced encryption standard AES on round 1. [Online]. Available: http://csrc.nist.gov/archive/aes/index.html.

[17] W. Stallings, *Cryptography and Network Security*, 5th ed. Boston, MA: Pearson Education India, 2011.

[18] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. New York City, NY: Springer, 2002.

[19] C. Cid, S. Murphy, and M. Robshaw, *Algebraic Aspects of the Advanced Encryption Standard*. Springer New York, 2006.

[20] ——, "Small scale variants of the AES," in *Fast Software Encryption*, vol. 3557. Springer, 2005, pp. 145–162.

[21] N. Ferguson, R. Schroeppel, and D. Whiting, "A simple algebraic representation of rijndael," in *Selected Areas in Cryptography*. Springer, 2001, pp. 103–111.

[22] N. Courtois, A. Klimov, J. Patarin, and A. Shamir, "Efficient algorithms for solving overdefined systems of multivariate polynomial equations," in *Advances in Cryptology—EUROCRYPT 2000*. Springer, 2000, pp. 392–407.

[23] A. Kipnis and A. Shamir, "Cryptanalysis of the HFE public key cryptosystem by relinearization," in *Advances in cryptology—CRYPTO'99*. Springer, 1999, pp. 788–788.

[24] N. T. Courtois and J. Pieprzyk, "Cryptanalysis of block ciphers with overdefined systems of equations," in *Advances in Cryptology—ASIACRYPT 2002*. Springer, 2002, pp. 267–287.

[25] J. C. Faugère, "A new efficient algorithm for computing gröbner bases (F4)," *Journal of Pure and Applied Algebra*, vol. 139, no. 1, pp. 61–88, 1999.

[26] ——, "A new efficient algorithm for computing gröbner bases without reduction to zero (F5)," in *Proc. of the 2002 international symposium on Symbolic and algebraic computation*. ACM, 2002, pp. 75–83.

[27] A. Kaminsky, M. Kurdziel, and S. Radziszowski, "An overview of cryptanalysis research for the advanced encryption standard," in *MILITARY COMMUNICATIONS CONFERENCE, 2010-MILCOM 2010*. IEEE, 2010, pp. 1310–1316.

[28] A. Bogdanov, D. Khovratovich, and C. Rechberger, "Biclique cryptanalysis of the full AES." Springer, 2011, pp. 344–371.

[29] T. E. Schilling and H. Raddum, "Solving equation systems by agreeing and learning," in *Arithmetic of Finite Fields*. Springer, 2010, pp. 151–165.

[30] K. Matheis, "An algebraic attack on block ciphers," Ph.D. dissertation, Flordia Atlantic University, 2010.

[31] S. Simmons, "Algebraic cryptanalysis of simplified AES," *Cryptologia*, vol. 33, no. 4, pp. 305–314, 2009.

[32] E. Kleiman, "High performance computing techniques for attacking reduced version of AES using XL and XSL methods," Ph.D. dissertation, Iowa State University, 2010.

[33] S. Bulygin and M. Brickenstein, "Obtaining and solving systems of equations in key variables only for the small variants of AES," *Mathematics in Computer Science*, vol. 3, no. 2, pp. 185–200, 2010.

[34] M. Brickenstein and A. Dreyer, "PolyBoRi: A framework for gröbner-basis computations with boolean polynomials," *Journal of Symbolic Computation*, vol. 44, no. 9, pp. 1326–1345, 2009.

[35] N. AlFardan and K. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," 2013.

[36] A. Greenberg. Cryptographers demonstrate new crack for common web encryption. [Online]. Available: http://www.forbes.com/sites/andygreenberg/2013/03/13/cryptographers-show-mathematically-crackable-flaws-in-common-web-encryption/.

[37] J. Daemen, L. Knudsen, and V. Rijmen, "The block cipher Square," in *Fast Software Encryption*. Springer, 1997, pp. 149–165.

[38] ——, "Implementation of square," 1997. [Online]. Available: http://www.esat.kuleuven.ac.be/~rijmen/square.

[39] V. Rijmen, J. Daemen, B. Preneel, A. Bosselaers, and E. De Win, "The cipher SHARK," in *Fast Software Encryption*. Springer, 1996, pp. 99–111.

[40] P. S. L. M. Barreto. The anubis block cipher. [Online]. Available: http://www.larc.usp.br/~pbarreto/AnubisPage.html.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia
2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California