NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

COMPUTATIONAL ALGEBRAIC ATTACKS ON THE ADVANCED ENCRYPTION STANDARD (AES)

by

Mantzouris Panteleimon

September 2009

Thesis Advisor:	David Canright
Co-Advisor:	Jon Butler

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Computational Algebraic Attacks on the Advanced Encryption Standard (AES)		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Mantzouris Panteleimon			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. 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.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

This thesis examines the vulnerability of the Advanced Encryption Standard (AES) to algebraic attacks. It will explore how strong the Rijndael algorithm must be in order to secure important federal information.

There are several algebraic methods of attack that can be used to break a specific cipher, such as Buchburger's and Faugere's F_4 and F_5 methods. The method to be used and evaluated in this thesis is the Multiple Right Hand Sides (MRHS) Linear Equations. MRHS is a new method that allows computations to be more efficient and the equations to be more compact in comparison with the previously referred methods.

Because of the high complexity of the Rijndael algorithm, the purpose of this thesis is to investigate the results of an MRHS attack in a small-scale variant of the AES, since it is impossible to break the actual algorithm by using only the existent knowledge. Instead of the original ten rounds of AES algorithm, variants of up to four rounds were used.

Simple examples of deciphering some ciphertexts are presented for different variants of the AES, and the new attack method of MRHS linear equations is compared with the other older methods.

This method is more effective timewise than the other older methods, but, in some cases, some systems cannot be uniquely solved.

14. SUBJECT TERMS Advanced Encryption Standard (AES), Rijndael's algorithm, Block Cipher, Decipher, Round of the algorithm, Sparse Multivariate Polynomial			**15. NUMBER OF PAGES** 121
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

COMPUTATIONAL ALGEBRAIC ATTACKS ON THE ADVANCED ENCRYPTION STANDARD (AES)

Panteleimon Mantzouris
Lieutenant Junior Grade, Hellenic Navy
B.S., Hellenic Naval Academy, 2001

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

and

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
September 2009

Author: LTJG Panteleimon Mantzouris

Approved by: David Canright
Thesis Advisor

Jon Butler
Co-Advisor

Jeffrey Knorr
Chairman, Department of Electrical and Computer Engineering

Carlos Borges
Chairman, Department of Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis examines the vulnerability of the Advanced Encryption Standard (AES) to algebraic attacks. It will explore how strong the Rijndael algorithm must be in order to secure important federal information.

There are several algebraic methods of attack that can be used to break a specific cipher, such as Buchburger's and Faugere's F_4 and F_5 methods. The method to be used and evaluated in this thesis is the Multiple Right Hand Sides (MRHS) Linear Equations. MRHS is a new method that allows computations to be more efficient and the equations to be more compact in comparison with the previously referred methods.

Because of the high complexity of the Rijndael algorithm, the purpose of this thesis is to investigate the results of an MRHS attack in a small-scale variant of the AES, since it is impossible to break the actual algorithm by using only the existent knowledge. Instead of the original ten rounds of AES algorithm, variants of up to four rounds were used.

Simple examples of deciphering some ciphertexts are presented for different variants of the AES, and the new attack method of MRHS linear equations is compared with the other older methods.

This method is more effective timewise than the other older methods, but, in some cases, some systems cannot be uniquely solved.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

This thesis compares different algebraic methods for solving and breaking the Advanced Encryption Standard (AES) algorithm. However, emphasis is given to the Multiple Right Hand Side (MRHS) Linear Equation method, which is a new algebraic method used for attacking ciphers with specific algebraic structures. Such a cipher is the Rijndael algorithm, which was adopted by the U.S. government as the most efficient among others in order to secure federal information.

In cryptography, it is considered that a cipher is broken when a method other than the brute force attack can reduce the complexity of the algorithm or completely decipher it. The “brute force attack” method is a very simple method in which we try one by one all the possible keys in order to break the cipher. In particular, due to the complexity of the AES algorithm, it becomes very difficult to break it with this method. Therefore, AES generated interest to find a specific algorithm to break this cipher. Some of the older algebraic methods, such as Buchberger’s algorithm and Faugere’s F_4 and F_5 algorithms, which are briefly presented in Chapter II, cannot break this cipher. This work considers a new algebraic method (MRHS) that can break small variants of the AES. Even the breaking of a small variant of the algorithm constitutes a success, since it opens a new horizon in the cryptanalysis field.

A comparison with the older algebraic attack methods that are presented in [9] shows that the MRHS method is faster than the others. In Chapter II, there is a quick overview for some of these methods and a more detailed analysis of MRHS method with some algebraic examples.

The overall concept of this thesis is to create a program that will break a small variant of the Rijndael algorithm. This work is based on the codes that Professors Håvard Raddum and Igor Semaev (of University of Bergen, Norway) provided to us. These codes (in the “C” language) are the application of the

MRHS method to the AES algorithm. In this thesis, we created new programs, which construct the equation systems of the AES algorithm, that we tried to solve by using the codes of Professors Raddum and Semaev. Our computational experiments examined many cases never considered in previous work.

This thesis demonstrates that, for a small variant of the AES, this method is very efficient (few high-level operations). In the third chapter, some examples are contained that show how this algorithm works. In some of the computational experiments, a totally new and significant result appeared for the cryptanalysis field; sometimes a small AES variant can have multiple decryption keys for a specific encryption key. This result is surprising and is not mentioned anywhere by Professors Raddum and Semaev.

In particular, the first example shows a case in which the MRHS method finds multiple solutions to the system (it fails to solve the system—break the cipher) and in the second one, all the MRHS linear equations are transformed into ordinary linear equations whose total number is the number of variables in our system.

ACKNOWLEDGMENTS

I would like to thank Professors Håvard Raddum and Igor Semaev, who very generously sent their codes with instructions and offered to help us.

In addition, I want to thank my advisor, Professor David Canright, for his invaluable help, without which the completion of this thesis would have been impossible.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. HISTORY

In 1976, the United States government as an official Federal Information Processing Standard (FIPS) adopted the Data Encryption Standard (DES). Today, this standard is considered insecure, and the necessity for a new more secure and efficient standard has emerged. Therefore, in 1997, the National Institute of Standards and Technology published a call for the replacement of DES. Among the requirements this new system had to meet were: the new algorithm should be able to allow key sizes of 128, 192 and 256 bits; it should operate on blocks of 128 input bits, and it should work in a variety of different hardware. For example, 8-bit processors that could be used in smart cards and the 32-bit architecture commonly used in personal computers [1].

For the adoption of the new algorithm, five finalists were chosen: MARS (from IBM), RC6 (from RSA Laboratories), Rijndael (from Joan Daemen and Vincent Rijmen), Serpent (from Ross Anderson, Eli Biham and Lars Knudsen) and Twofish (from Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson). Finally, in 2001, the Rijndael algorithm [2] was adopted as the Advanced Encryption Standard (AES) [3].

B. RELATED WORK

This new algorithm was of great interest for the area of cryptanalysis. The cryptanalists started to seek ways to attack this algorithm, which might lead to breaking the algorithm (finding the key more efficiently than a brute-force attack). Since Rijndael's algorithm is defined using algebraic operations in finite fields, cryptanalists investigated algebraic methods in order to break it. Therefore, they started to try the already known algebraic techniques for solving systems of polynomial equations. Some of them were the Buchburger's algorithm and the F_4 and F_5 algorithms [4] that are summarized in Chapter II with some simple

examples. Also, they tried to describe the new algorithm by using some different algebraic representations over the Galois finite fields [4] that are discussed in the same chapter.

Recently, a new algebraic method was developed that used Multiple Right Hand Sides (MRHS) Linear Equations [5]. The main difference with the other algebraic methods is that the equations, which describe the algorithm, are not expressed anymore as multivariate polynomial equations, but instead they are presented as a system of linear equations, each having a set of multiple right hand sides.

The Norwegian Professors Håvard Raddum and Igor Semaev first developed this method. After a comparison between the Buchberger's F_4 algorithms and the MRHS method, they concluded that the last one is much faster.

In this thesis, after we borrowed the codes [6] from the Norwegian team, we managed to simulate the AES algorithm in the C language and to break small-scale variants of the Rijndael's algorithm. After the application of the above-mentioned codes, many computational experiments were executed to explore how the method would work in many different cases. Though the previous work of Raddum and Semaev [5] only considered AES variants using the 8-bit field, we also explored variants based on 4-bit and 2-bit fields. From these experiments, we obtained some very surprising new results about the decryption (breaking) of variants of the Rijndael algorithm. In particular, one of the them is that there are some cases that a ciphertext, which was encrypted by a specific key, could be broken by using a definite number of different keys. This is explained in detail in Chapter III.

C. INTRODUCTION TO AES

Although the Rijndael algorithm was adopted as the official algorithm for AES, it is important to note that the algorithm, which is used in the AES, is a limited version of the Rijndael's algorithm. In particular, Rijndael is a block cipher with both a variable block length and a variable key length [2]. While Rijndael's algorithm can be used for block and key lengths in any multiple of 32 bits, with a minimum of 128 bits and a maximum of 256 bits, AES uses constant block length of 128 bits and key lengths of 128, 192 and 256 bits only. Also, this algorithm is applicable with a different number of rounds (N_r), depending on the number of columns of the cipher key (N_k) and the number of columns of the rectangular array (N_b) called state, which is actually the intermediate cipher result of the algorithm. In Table 1, the number of rounds are shown as a function of the block and key length.

	N_b				
N_k	4	5	6	7	8
4	10	11	12	13	14
5	11	11	12	13	14
6	12	12	12	13	14
7	13	13	13	13	14
8	14	14	14	14	14

Table 1. Number of rounds, N_r, as a function of N_b (block length/32) and N_k (key length/32).

Since, in this thesis, only the AES algorithm will be examined, it is good to analyze the subcases of the Rijndael's algorithm that are used in the AES. For the encryption process, the input is the plaintext block and the initial key and the

output is the ciphertext block. In the decryption process, the input is the ciphertext block and the output the plaintext block.

With a 128-bit key length, the algorithm applies 10 rounds. (A 192-bit key length contains 12 rounds and a 256-bit length contains 14 rounds). In each round, there are four transformations (linear and non-linear) that are also called layers. Each round has also a round key, derived from the original key (input key). The round transformation and its steps generate intermediate data called states. A state can be considered as a rectangular array of bytes with four rows and a number of columns (N_b) that depend on the size of the key length. Here, we consider that the key length is 128 bits, where the key is arranged in a 4x4 matrix such that each element is a byte. The four transformations are the ByteSub transformation, the ShiftRow transformation, the MixColumn transformation and the AddRoundKey transformation. These four transformations compose a round. The four transformations are discussed next.

D. STRUCTURE OF THE AES ALGORITHM

The AES algorithm can be separated in two stages—the encryption and the decryption process. The algorithm for the encryption process includes four transformations, as it is shown below. The algorithm for the decryption process consists of the inverses of the above-mentioned transformations in the reverse order.

1. Encryption Process

The sequence of the four transformations mentioned above is the following: ByteSub, ShiftRow, MixColumn and AddRoundKey transformations. We next present these transformations in order.

a. The ByteSub Transformation (BS)

This is the only non-linear part of the algorithm and assures resistance to differential and linear cryptanalysis attacks [2]. This transformation

consists of an S-box, which is applied to each byte element of the state (16-byte block) independently and has three different steps: inversion, a Galois Field (GF) linear mapping, and S-Box constant ,as it is shown in Figure 1.

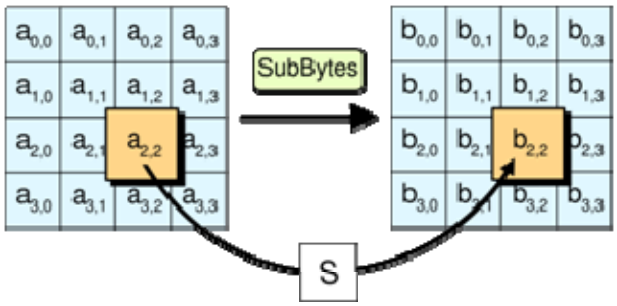

Figure 1. The ByteSub step, the first stage in a round of AES (From [7]).

(1) Inversion. In this operation of the S-box, the inverse is computed in the 8-bit Galois Field, GF(2^8). The byte 00000000 has no inverse and 00000000 is used in place of its inverse. Assume that the first byte is $x_7x_6x_5x_4x_3x_2x_1x_0$. The byte, which comes up from the inversion, will be $y_7y_6y_5y_4y_3y_2y_1y_0$, which represents an eight element column vector, with the rightmost binary bit y_0 in the top position. This operation provides resistance against the linear and differential cryptanalysis attacks [1].

(2) GF—Linear Mapping. At this point, the y vector is multiplied by a constant matrix, and the column vector (0,1,1,0,0,0,1,1) is added, yielding a vector $z_7z_6z_5z_4z_3z_2z_1z_0$: (Note: Galois addition is equivalent to bitwise XOR in any finite field of even size).

$$\begin{bmatrix} z_7 \\ z_6 \\ z_5 \\ z_4 \\ z_3 \\ z_2 \\ z_1 \\ z_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

(3) S-box Table. The byte z is the input to the S-box table. Consider an input byte abcdefgh. The S-box is a 16x16 matrix. We look for the entry at abcd row and efgh column (rows and columns are numbered from 0 to 15). The intersection of these two entries, transformed into a binary number, is the output from the S-box. For example, if the figure 10101010 is the input byte, the first four bits, 1010, represent the decimal 10. Therefore, one enters at the eleventh row and eleventh column. The intersection is 172, as it is shown in Figure 2. This is converted into binary, which is 10101100. This is shown as 'ac,' which is hexadecimal for 10101100 in Table 2. That number is the output of the S-box.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 2. S-Box in the encryption process (From [3]).

In order to make clear the hexadecimal notation, Table 2 shows the correspondence between decimal, binary and hexadecimal notations.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 2. Correspondence between decimal, binary and hexadecimal notations.

b. The ShiftRows Transformation (SR)

In this transformation, which is linear, the rows of each state are cyclically shifted to the left, with each row shifted a different amount, as it appears in Figure 3. This provides resistance against truncated differential and saturation attacks [2]. For example, row zero is shifted by C_0 bytes, row 1 is shifted by C_1 bytes, row 2 is shifted by C_2 bytes in such a way that the byte at position j in row i moves to position (j-C_i) mod N_b. Particularly in AES $C_0 = 0$, $C_1 = 1$, $C_2 = 2$ and $C_3 = 3$. The output of this transformation is a matrix with the same dimensions as the input matrix.

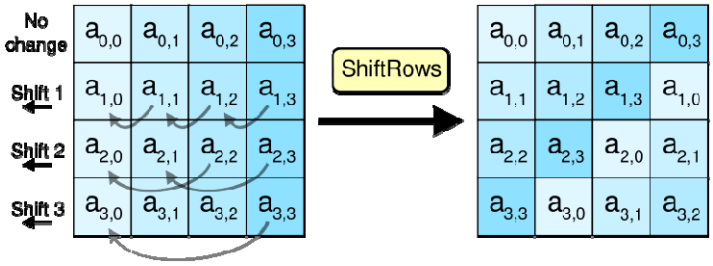

Figure 3. The `ShiftRows` step, the second stage in a round of AES (From [7]).

c. The MixColumns Transformation (MC)

This transformation operates on each 4-byte column separately and is omitted in the last round. It is also a linear transformation, which has diffusion power. The columns of the state are considered as polynomials over GF(2^8), which are multiplied by a fixed polynomial c(x) modulo (x^4+1). This polynomial is:

$$c(x) = 03 \cdot x^3 + 01 \cdot x^2 + 01 \cdot x + 02 \quad [2]$$

and this multiplication can be presented as a matrix multiplication [2] (in hexadecimal notation), as it appears below:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad [2]$$

Figure 4 depicts the MixColumns operation on the columns of the state.

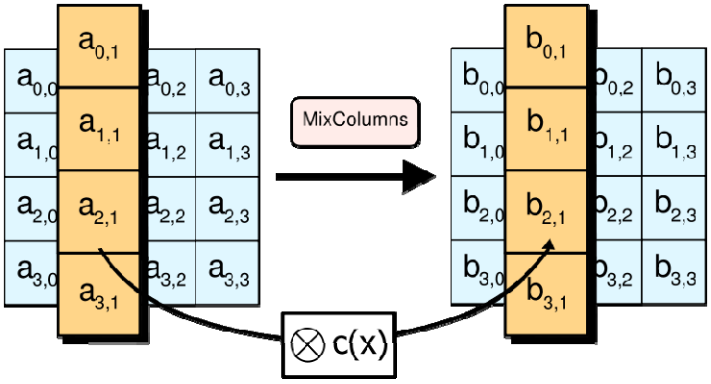

Figure 4. The MixColumns step, the third stage in a round of AES (From [7]).

d. The AddRoundKey Transformation

In this transformation, a key, consisting of 128 bits, which are arranged in a 4x4 byte matrix, is added to the output of the MixColumn transformation. A different round key is added to the state at the end of each round.

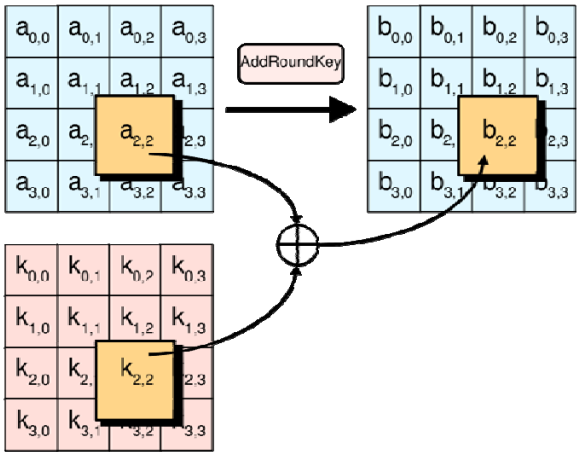

Figure 5. The AddRoundKey step, the fourth stage in a round of AES (From [7]).

This key is derived recursively from the original key as follows. We will see this procedure in five steps.

Step1: Label the first four columns of the original key W(0), W(1), W(2), W(3).

Since the whole algorithm consists of 10 rounds, 40 more columns are required, four for each round. Let i symbolizes the number of column in the different round keys as the columns are derived from the key schedule ($4 \le i \le 43$).

Step 2: If the number of the new column is a multiple of four, then…

Step 3: It is $W(i) = W(i-4) \oplus T(W(i-1))$, where $T(W(i-1))$ is the transformation of $W(i-1)$ obtained as follows. Assume that the elements of the column $W(i-1)$ are a,b,c,d. These are shifted cyclically to obtain b,c,d,a, and then, at this point, each of these bytes is substituted with its corresponding byte from the S-box of the ByteSub transformation, as it is explained above. So, four other bytes result, i.e., e,f,g,h.

Step 4: Finally, the round constant

$$r(i) = 00000010^{(i-4)/4}$$

is computed in GF(2^8). So, the $T(W(i-1))$ is the column vector $(e \oplus r(i), f, g, h)$.

Step 5: If i is not a multiple of four, then

$$W(i) = W(i-4) \oplus W(i-1).$$

2. Decryption Process

The decryption process consists of the inverses of the four encryption steps: InvByteSub, InvShiftRows, InvMixColumns and invAddRoundkey.

a. The InvByteSub Transformation

This transformation consists of the inverse S-box. In essence, in this step, the inverse transformation of the equation that was made in the ByteSub transformation is performed. For the linear mapping, one takes:

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix}$$

The inverse of the S-box appears in Figure 6:

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 6. S-Box in the decryption process (From [3]).

b. The InvShiftRows Transformation

In this transformation, the opposite shifting operation is applied. Therefore, the rows are shifted to the right instead of to the left, which takes place at the ShiftRows transformation.

c. The InvMixColumns Transformation

In this transformation, every column is multiplied by the inverse polynomial of c(x) (mod x^4+1) which is:

$$d(x) = 0B \cdot x^3 + 0D \cdot x^2 + 09 \cdot x + 0E$$

The inverse matrix multiplication of the equation, which was used in the MixColumn transformation, is:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

This transformation is omitted in the last round.

d. The InvAddRoundKey Transformation

This transformation applies the keys that were used in the encryption process in the reverse order.

E. THESIS OBJECTIVE

A common method to break a cipher is called “the brute force attack.” In this method, one tries to break a cipher using one or more ciphertexts, where the corresponding plaintexts are known, by trying to decrypt them using all the possible keys for the cipher. When the decryption yields the known plaintext, we have found the key and the cipher is broken. (A variation on this method, used when the plaintexts are not known, stops when a meaninful plaintext is found).

However, as ciphers become more and more complicated, this method turned out to be useless. For example, in the case of the Rijndael's algorithm, one has to try at most 2^{128} possible keys to break the cipher. This is impractical, since the time and the memory that are necessary for that exceed the limits of all existing workstations.

In the last few decades, some cryptanalysis has been based on algebraic attacks. Algebraic attacks have the advantage that a cryptanalyst does not need to have many known plaintexts and ciphertexts in order to create an equation that can describe the cipher, an event that happens with the linear or differential cryptanalysis where one has to have many pairs of plaintexts and ciphertexts in order to be able to describe the cipher.

In this thesis, a specific algebraic attack method—the Multiple Right Hand Side (MRHS) Linear Equations method—is examined and compared with other older methods (Buchburgers, F_4, F_5) [4]. In addition, a small-scale variant of the AES will be examined, since, as it will be explained later, the number of the variables for the whole AES algorithm is too big to be solved.

Therefore, in Chapter II we will briefly summarize these older algebraic methods. While these methods can represent the AES, they cannot break the cipher. A more detailed analysis of the MRHS Linear Equations method will also be presented with a very simple example in order to show how this method works.

In Chapter III, we will present the computational experiments, where we apply the MRHS approach to small variants of AES, including several that have never been attacked before by this method. We present the results of the application of this method with two representative examples. In the first example, the method fails to solve the algebraic system that is created with the MRHS Linear Equations, while in the second one we conclude to a solution.

THIS PAGE INTENTIONALLY LEFT BLANK

II. ALGEBRAIC EQUATIONS FOR AES

A. INTRODUCTION

There are many different ways to describe a cipher. However, the complexity of modern ciphers requires knowledge of their algebraic representation in order to attack and break them. In particular, one tries to describe a cipher by finding the algebraic properties that it has and, after that, by creating some homomorphisms or isomorphisms of this cipher. These new structures are very helpful, both to the implementers of a particular cipher who want to provide further protection to the cipher against side-channel attacks, and to the cryptanalysts who try to analyze it or even to break it.

B. DIFFERENT REPRESENTATIONS OVER F_2 AND F_{256}

The meaning of F_2 is the finite field with order (number of elements) $p^n = 2^1$, where p is always a prime number and n is a positive integer. Respectively, F_{256} is the finite field with order $p^n = 2^8 = 256$, which is also notated GF(2^8).

After the adoption of the modern algorithms, one of which is the Rijndael's algorithm, a great interest was created for the wider application of computational algebra in cryptography. Therefore, a number of different algebraic representations were developed.

In the case of AES, one can consider a number of different representations, which are called dual ciphers depending on the properties of their representation mappings. Some are described in the next pages.

1. Isomorphic Representations

Definition: *Suppose* A *is a vector space over a field* F *with a multiplication operation* $A \times A \to A$. *If this multiplication operation is associative*

and is a bilinear mapping on the vector space A*, then* A *is an (associative)* F*-algebra, or, more simply, an algebra*[4].

In these ciphers, the mappings of the state and key spaces are algebra isomorphisms of the AES state space algebra, where algebra is defined below. Therefore, these ciphers are isomorphic to the AES.

In AES, each byte can be considered as an element of the finite field $GF\left(2^8\right)$ in terms of the following polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \text{ (Rijndael polynomial)}$$

which is irreducible in $GF(2)[x]$. This finite field can be constructed in many different ways from the chain of its subfields, as it appears below:

$$GF(2) \subset GF\left(2^2\right) \subset GF\left(2^4\right) \subset GF\left(2^8\right)$$

In total, there are $30 + (1 \cdot 60) + (3 \cdot 120) + (1 \cdot 6 \cdot 120) = 1170$ different isomorphic representations of the AES based on the different irreducible polynomials of the $GF\left(2^8\right)$ subfields as they appear in Table 3 [4].

	Subfield		
Degree	GF (2)	GF (2^2)	GF (2^4)
2	1	6	120
4	3	60	-
8	30	-	-

Table 3. The number of irreducible polynomials over subfields of GF (2^8).

These particular representations are not of cryptanalytic interest. Rather, they are intended to improve the efficiency of hardware implementation.

2. Regular Representations

The regular representation is the algebra homomorphism $v : A \to M_n(K)$ that maps $a \in A$ to the matrix corresponding to the linear transformation $z \mapsto az$, where z is a vector over F of length n [4].

3. Logarithmic Representations

Since an element of a finite field can be represented with logarithmic functions instead of vector spaces, an element of the AES state space (excluding zero bytes) can be described as an element of the set $(\mathbb{Z}_{255})^{16}$. These ciphers are called log dual ciphers and there are 128 different primitive elements in F, giving 128 such representations [4]. More details of how to specify a logarithmic representation of the AES are given in [8].

C. ALGEBRAIC SOLUTION METHODS

In this subsection, some algebraic solution methods will be discussed. The first two will be based on the Gröbner basis algorithms, which are well-known methods for the solution of multivariate polynomial equations. These two algorithms have been unable to break the Rijndael algorithm, and at the end of this chapter there will be a comparison between these algorithms and the MRHS Linear Equation algorithm in terms of their complexities [4]. The last one, which is examined and analyzed in this thesis with more details, is a new approach that seems to have better results than the previous two based on the time that is needed in order to break the AES cipher.

1. Buchberger's Algorithm

As mentioned above, Buchberger's algorithm solves systems of multivariate polynomial equations. Consider a polynomial ring $\mathbb{F}[x_1, x_2, ..., x_n]$ with

a monomial ordering. Suppose $I \lhd \mathbb{F}[x_1, x_2, ..., x_n]$ is an ideal of this ring with a basis $F = \{f_1, ..., f_m\}$. The S-polynomial of any pair of the function of the basis is defined as:

$$S(f_i, f_j) = \left(\frac{lcm(LM(f_i), LM(f_j))}{LT(f_i)} \right) f_i - \left(\frac{lcm(LM(f_i), LM(f_j))}{LT(f_j)} \right) f_j,$$

where lcm is the *least common multiple,* LM is the *Leading Monomial,* and LT is the *Leading Term*. What one tries to achieve with this polynomial S, which belongs to the ideal I, is to cancel the leading terms from any pair of the polynomials and, with the help of the next theorem, to compute a Gröbner basis of the ideal I [4].

Theorem: *Let $\mathbb{F}[x_1, x_2, ..., x_n]$ be a polynomial ring with a monomial ordering, and let I be an ideal of $\mathbb{F}[x_1, x_2, ..., x_n]$. A basis $G = \{f_1, ..., f_m\}$ for the ideal I is a Gröbner basis for I if and only if every S-polynomial $S(f_i, f_j)$ of pairs of distinct polynomials $f_i, f_j \in G$ has remainder 0 upon division by G.*

Consider an example of how to solve a multivariate polynomial system by using this algorithm.

Example: Consider a polynomial ring F[x,y] with multivariate polynomials (with two variables) over the complex numbers with the lexicographic order $(y \prec x)$. The ideal in this particular example is generated from the polynomials that appear below:

$$f_1 = x^2 y - 1 \text{ and } f_2 = xy^2 - x$$

Here, the Gröbner basis is computed by using the Buchburger's algorithm. Set $G = \{f_1, f_2\}$ and compute the $S(f_1, f_2)$-polynomial.

$$S(f_1, f_2) = \frac{x^2 y^2}{x^2 y}(x^2 y - 1) - \frac{x^2 y^2}{xy^2}(xy^2 - x)$$

$$= y\left(x^2 y - 1\right) - x\left(xy^2 - x\right) = x^2 - y$$

The leading term of the $S(f_1, f_2)$ is x^2. The leading term of f_1 and f_2 is $x^2 y$ and xy^2, respectively. So, $S(f_1, f_2)$ cannot be reduced by these two polynomials. The G is expanded, so that it becomes $G = \{f_1, f_2, f_3\}$ where $f_3 = x^2 - y$ and $S(f_1, f_3)$ and $S(f_2, f_3)$ are computed.

$S(f_1, f_3) = y^2 - 1$, which cannot be reduced by the set $G = \{f_1, f_2, f_3\}$. So the set $G = \{f_1, f_2, f_3, f_4\}$ is expanded, where $f_4 = y^2 - 1$.

$S(f_2, f_3) = -x^2 + y^3$. Its leading term is $-x^2$ which is divisible by the $LT(f_3)$. The division of $S(f_2, f_3)$ by f_3 and then by f_4 results in:

$$S(f_2, f_3) = -x^2 + y^3 = -f_3 + \left(y^3 - y\right) = -f_3 + yf_4$$

Now, $S(f_1, f_4) = f_3$, $S(f_2, f_4) = 0$ and $S(f_3, f_4) = f_3 - yf_4$ are also computed. Thus, one concludes that all these S-polynomials can be reduced by G. So, $G = \{f_1, f_2, f_3, f_4\}$ is a Gröbner basis of the ideal I. The reduced Gröbner basis is $G = \left\{x^2 - y, y^2 - 1\right\}$. Finally, the following system of equations is solved in order to find the solution.

$$x^2 - y = 0$$
$$y^2 - 1 = 0$$

The complete solution of this system is $\{(1,1), (-1,1), (i,-1), (-i,-1)\}$.

2. F_4 and F_5 Algorithms

These two algorithms have their names from their creator, Jean Charles Faugere, and compute again a Gröbner basis. They can be viewed as an improved method of Buchburger's algorithm. They are based on the same principles. In particular, the F_4 algorithm instead of polynomial reduction uses

matrix reduction, while F_5 uses matrix reduction, but each of these generated matrices is of full rank. This method is illustrated in the next example [4].

Example: Consider the polynomial ring $R[x, y, z]$, which includes polynomials with three variables in lexicographic order. We want to reduce the following polynomials:

$$f_1 = 3x^3yz - 5xy \text{ and } f_2 = 5x^2z^2 + 3xy + 1$$

by the set of polynomials $\{g_1, g_2\}$, where

$$g_1 = xy - 2z \text{ and } g_2 = x^2z - 3yz .$$

To reduce f_1, with respect to $\{g_1, g_2\}$, the following reduction is performed:

$$\begin{aligned} f_1 &= 3x^3yz - 5xy \\ &= 6x^2z^2 - 5xy + \left(3x^2z\right) g_1 \\ &= -5xy + 18yz^2 + \left(3x^2z\right) g_1 + \left(6z\right) g_2 \\ &= 18yz^2 - 10z + \left(3x^2z\right) g_1 + \left(6z\right) g_2 - \left(5\right) g_1 \end{aligned}$$

So, f_1 is reduced with respect to $\{g_1, g_2\}$ to $18yz^2 - 10z$.

The same procedure for f_2 is

$$\begin{aligned} f_2 &= 5x^2z^2 + 3xy + 1 \\ &= 3xy + 15yz^2 + 1 + \left(5z\right) g_2 \\ &= 15yz^2 + 6z + 1 + \left(5z\right) g_2 + \left(3\right) g_1 \end{aligned}$$

Thus f_2 is reduced to $15yz^2 + 6z + 1$ with respect to $\{g_1, g_2\}$.

The idea behind the F_4 and F_5 algorithms is to make these reductions as a matrix reduction. However, the reduction for both f_1 and f_2 requires reduction only with respect to $\left(x^2z\right)g_1$, g_1, and $\left(z\right)g_2$. Therefore, a matrix of coefficients is created.

$$\begin{array}{c} \\ f_1 \\ f_2 \\ x^2zg_1 \\ 1g_1 \\ zg_2 \end{array} \begin{array}{c} \begin{array}{cccccc} x^3yz & x^2z^2 & yz^2 & xy & z & 1 \end{array} \\ \begin{pmatrix} 3 & 0 & 0 & -5 & 0 & 0 \\ 0 & 5 & 0 & -3 & 0 & 1 \\ 1 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 0 \\ 0 & 1 & -3 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

The reduction steps correspond to the row reduction of the first two rows, which represent the polynomials f_1, f_2 using the last three rows, which represent the $\left(x^2z\right)g_1, g_1, \left(z\right)g_2$ polynomials. So, as a result of the reduction, the following matrix is derived.

$$\begin{array}{c} \\ f_1 \\ f_2 \\ x^2zg_1 \\ 1g_1 \\ zg_2 \end{array} \begin{array}{c} \begin{array}{cccccc} x^3yz & x^2z^2 & yz^2 & xy & z & 1 \end{array} \\ \begin{pmatrix} 0 & 0 & 18 & 0 & -10 & 0 \\ 0 & 0 & 15 & 0 & 6 & 1 \\ 1 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 0 \\ 0 & 1 & -3 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

The first two rows of the reduced matrix give the reduction of f_1, f_2 with respect to g_1, g_2. Note, that this is the same result that was achieved with the Buchburger's reduction.

3. Multiple Right Hand Sides (MRHS) Linear Equations Algorithm

A different approach of algebraic attack not based on equations of polynomials will be examined. This presentation will utilize a special type of equation known as Multiple Right Hand Side Linear Equations.

Definitions:

1. A set of small-scale Variants of the AES denoted by *SR(n,r,c,e)* are defined, where

- n is the number of rounds,
- r is the number of rows in the rectangular arrangement of the input,
- c is the number of columns in the rectangular arrangement of the output,
- e is the number of bits in a "word."

Since we consider that *e=8,* the underlying finite field is $GF(2^8)$ and all the matrices and vectors are over *GF(2)*.

2. Let *X* be a set of Boolean variables represented as a column vector. An equation of the form

$$AX = a_1, a_2, ..., a_s$$

or

$$S_i : A_i X = [L_i]$$

is called an **MRHS system of linear equations** if *A* is a matrix of size *k*x*n* and rank *k* and $a_1, a_2, ..., a_s$ are column vectors of length *k*.

3. A symbol *S=(X,L)* consists of an ordered set of variables *X=X(S)* and a list *L=L(S)* of Right Hand Sides [9]

a. Agreeing Procedure

One of the problems with applying algebraic attacks in a cipher is the difficulty of presenting practical examples, because the required time and memory requirements grow beyond the limitations of a typical workstation. Therefore, only small-scale variants of the different ciphers were performed.

After the completion of the rounds of AES, one can count the number of variables there are in the system in order to determine the algebraic equations that can describe the system.

For example, the total number of variables for the AES, assuming 8-bit words, is given by the following formula:

$$8rc + 8nr + 8(n-1)rc$$

Thus, for a complete AES the number of variables is 1600. This is large, if one wants to solve a system with so many non-linear polynomial equations.

Figure 7 shows the bytes that are variables in a small-scale variant of the AES.

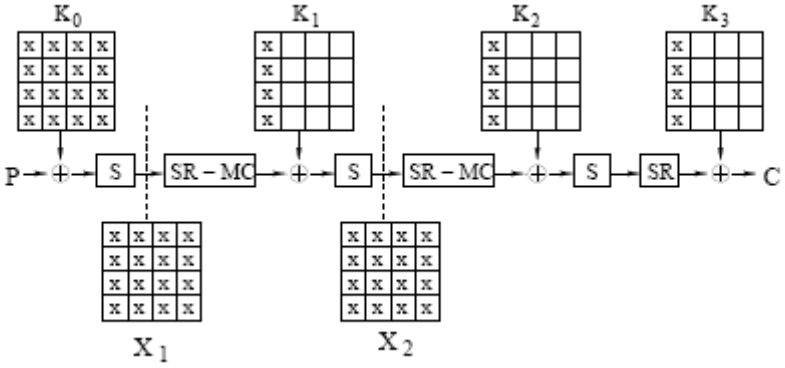

Figure 7. SR(3,4,4,8) equations variables (From [9]).

The variables can be separated in two different categories. The first one includes all the bits that are derived after each S-box, except the last one that is considered as known, since the ciphertext is considered as a known for this algorithm. The second one is the 16 bytes of the initial key and after that, only the first column of each expansion key, since the first column is derived with the contribution of the S-box, while the three others are dependent on the first column, as it is shown in Figure 7 and described in the AddRoundKey transformation in Chapter I.

Consider two symbols:

$$S_i : A_i X = [L_i] \qquad \text{and} \qquad S_j : A_j X = [L_j]$$

The matrices L_i are of size $k_i x s_i$. These symbols are derived from the bits that go in and out of each S-box, which are expressed as linear combinations of the variables.

These two symbols agree if, for any $a_1 \in L_i$, there exists an $a_2 \in L_j$ such that the linear system

$$\begin{bmatrix} A_i \\ A_j \end{bmatrix} X = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

is consistent, and, conversely, for any $a_2 \in L_j$, there exists an $a_1 \in L_i$ such that the linear system is consistent. The steps for the application of this attack are as follows:

- Define the matrices:

  $$A = \begin{bmatrix} A_i \\ A_j \end{bmatrix}, T_{ij} = \begin{bmatrix} L_i \\ 0 \end{bmatrix}, T_{ji} = \begin{bmatrix} 0 \\ L_j \end{bmatrix} \text{ with } t = k_i + k_j \text{ rows.}$$

- Choose a nonsingular transform matrix U of size $t \times t$ such that the product UA is a matrix with zeros in its last r rows. If $r=0$, then the symbols agree.
- If $r>0$, then compute the matrices UT_{ij} and UT_{ji}. Let Pr_{ij} denotes the set of UT_{ij}-column projections to the last r coordinates. If $\mathrm{Pr}_{ij} = \mathrm{Pr}_{ji}$ the symbols agree.
- If $\mathrm{Pr}_{ij} \neq \mathrm{Pr}_{ji}$ then remove the columns from L_i whose image is not found in $\mathrm{Pr}_{ij} \cap \mathrm{Pr}_{ji}$ and, similarly, the columns from L_j whose image is not found in $\mathrm{Pr}_{ij} \cap \mathrm{Pr}_{ji}$. One concludes with two new symbols whose equations are $S_i^{'} : A_i X = [L_i^{'}]$ and $S_j^{'} : A_j X = [L_j^{'}]$.

This procedure is repeated up to the point that all symbols agree. When the agreeing procedure is applied in a pair of symbols S_i and S_j and the procedure is

continued with other symbols in the system, for example, S_k, it is possible that the S_i and S_k will disagree. This means that S_i and S_j may disagree again after S_i and S_k agree. So, the previous agreements have to be run through again and again.

b. Gluing Procedure

After the completion of the agreeing procedure, one may conclude that the system of equations does not have a solution. Therefore, a way to start the agreeing procedure again must be found. Here is where the *gluing procedure* comes into play. This method merges two symbols into one, bringing their joint information in the new symbol. The agreeing procedure is applied again with the new symbols, and is repeated until a unique solution is reached. The following example demonstrates how this works. Also, this step increases the complexity of the attack.

c. Example

1. Agreeing Procedure. We have two equations $A_1X = [L_1]$ and $A_2X = [L_2]$ in variables $X = \{x_1, x_2, x_3, x_4, x_5\}$:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad , \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

By following the previously described procedure, the matrix A is:

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

A non-singular matrix U for the transformation of A is:

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}.$$

So,

$$UA = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Then, r=2.

Put now

$$T_{12} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad T_{21} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

and compute:

$$UT_{12} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad UT_{21} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}.$$

Looking at the last two rows of the above matrices, one defines:

$$Pr_{12}=\{(1,1),(1,0),(0,1)\}, \qquad Pr_{21}=\{(1,1),(0,0),(0,1)\}$$

$$Pr_{12}\cap Pr_{21}=\{(1,1),(0,1)\}$$

One can observe that the second and the fourth column of UT_{12} do not match with any column of UT_{21}. Therefore, these columns are removed. Similarly, the second and the third column of UT_{21} should be removed for the same reason.

In addition, two new symbols result,

$$\begin{bmatrix}1&1&0&0&0\\1&0&1&0&0\\1&0&0&1&0\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\\x_4\\x_5\end{bmatrix}=\begin{bmatrix}1&0\\0&0\\0&1\end{bmatrix} \quad , \quad \begin{bmatrix}0&1&0&0&1\\0&0&1&0&1\\0&0&0&1&1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\\x_4\\x_5\end{bmatrix}=\begin{bmatrix}0&0\\1&0\\1&1\end{bmatrix}$$

that now agree.

2. Gluing Procedure. Let *B* be the sub-matrix of *UA* in its last *t-r* nonzero rows. The gluing of the two previously agreed symbols is *BX=[L]*, where each column of *L* is the sum of one column from UT_{12} and one from the UT_{21} with the same projection in its last r coordinates, reduced to the first *t-r* rows.

For the above example, the gluing procedure yields:

$$\begin{bmatrix}1&1&0&0&0\\1&0&1&0&0\\1&0&0&1&0\\0&1&0&0&1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\\x_4\\x_5\end{bmatrix}=\begin{bmatrix}1&0\\0&0\\0&1\\0&0\end{bmatrix}$$

This MRHS system of linear equations contains the information of the first two different symbols.

d. From MRHS to Linear Equations

In order to derive the ordinary linear equations (unique right hand side) from the MRHS linear equations, the L matrix is triangulated with a row transformation. An upper-triangular matrix with zeros in its last $r_1 \geq 0$ rows is sought, from where to take r_1 homogeneous equations. There may also be non-homogeneous equations with ones in the whole row of *L*. In the above example, the triangulation of L results in the following symbol:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix},$$

which is equivalent with the three following linear equations and the two initial MRHS linear equations [5]:

$$x_2 + x_4 = 1$$
$$x_1 + x_3 = 0$$
$$x_2 + x_5 = 0$$

4. Algorithms' Complexities and Comparison

This section compares the complexities of the previously described algorithms.

The Buchberger's algorithm time complexity is related to the total degree of all the intermediate polynomials that are generated by the algorithm. In essence, this algorithm can have double exponential complexity. In particular, considering the AES equation system over $GF\left(2^8\right)$, the complexity of Buchburger's algorithm is, at worst, single exponential [4].

The F_4 and F_5 algorithms are different approaches for the computation of a Gröbner basis. Since the Buchberger's algorithm involves polynomial reductions,

which take place sequentially, while F_4 and F_5 use matrix reductions, it seems logical that F_4 and F_5 will be faster. The F_5 algorithm is even better than the F_4 because it uses only full rank matrices.

The complexity of the MRHS algorithm is very difficult to specify since the different parameters that can be used, such as the number of rounds, the number of rows and columns, etc, make it very complicated. However, this algorithm seems to be much better than the previous ones, since the required time to solve equations from the AES is much smaller [5].

THIS PAGE INTENTIONALLY LEFT BLANK

III. COMPUTATIONAL EXPERIMENTS

A. METHODOLOGY

This chapter will explain how the MRHS method was applied by using different equations that were generated with the code shown in Appendix B. The parameters of this small-scale variant of AES were determined: the number of rounds ($0<N_r<10$), the number of the rows and the columns in the rectangular arrangement of the input ($0<N_b<4$), and the number of bits in each word (2, 4, 8). In addition, a random plaintext and an initial key were chosen, which have lengths that correspond to the number of columns and rows of the states and with the number of bits in the words as well. These texts are expressed in hexadecimal notation. After that, the MRHS Linear Equations method was used to try to solve this system.

Initially, the agreeing procedure was implemented to see if some linear equations could be extracted just by applying this procedure. If the number of linear equations that resulted was equal to the number of variables in the system, then the procedure would be stopped since there would be a unique solution for the system. If the number of linear equations extracted from the agreeing procedure was less than the number of variables, the gluing procedure was performed, in which the maximum number of Right Hand Sides (RHS) in the glued equations is specified. Then another attempt was made to extract some linear equations. In order to have a unique solution, it is necessary to have a sum of linear equations equal to the number of variables.

B. RESULTS

Some simple examples of the procedure described above are shown below.

1. Example 1

In this example, a system was created with two rounds, two rows and two columns, in the rectangular arrangement of the input and 2-bit words. The system has also a plaintext **fa** (in hexadecimal notation), which corresponds to 11111010 in binary notation, and a key **ea** (in hexadecimal notation), which corresponds to 11101010 in binary notation. The system of equations appears below, starting with the number of bit variables and the number of MRHS equations. Each MRHS equation appears as the number of rows, the number of RHS, the rows of A_i and the columns of L_i. The first four symbols are from the key schedule, the next four from round 1, and the last four from round 2.

```
  24  12
 4 4
000000100000000000000000
000000010000000000000000
100000001000000000000000
010000000100000000000000
0011
0110
1000
1101
 4 4
000010000000000000000000
000001000000000000000000
001000000010000000000000
000100000001000000000000
0010
0111
1001
1100
 4 4
000000100010000000000000
000000010001000000000000
000000001000100000000000
000000000100010000000000
```

```
0000
0101
1011
1110
 4 4
00001000100000000000000
00000100010000000000000
00000000001000100000000
00000000000100010000000
0010
0111
1001
1100
 4 4
10000000000000000000000
01000000000000000000000
00000000000000010000000
00000000000000001000000
0010
0111
1001
1100
 4 4
00000010000000000000000
00000001000000000000000
00000000000000000100000
00000000000000000010000
0010
0111
1001
1100
 4 4
00001000000000000000000
00000100000000000000000
00000000000000000001000
00000000000000000000100
```

```
0010
0111
1001
1100
 4 4
00100000000000000000000
00010000000000000000000
00000000000000000000010
00000000000000000000001
0010
0111
1001
1100
 4 4
00000000100000000111000
00000000010000001110000
00000000000010000000000
00000000000001000000000
0001
0100
1010
1111
 4 4
00000010001000000001101
00000001000100000001011
00000000000000010000000
00000000000000001000000
0010
0111
1001
1100
 4 4
00001000100000000000111
00000100010000000001110
00001000100010000000000
00000100010001000000000
```

```
0010
0111
1001
1100
 4 4
000000000010000011010000
000000000001000010110000
000000100010001000000000
000000010001000100000000
0000
0101
1011
1110
now: neq = 12 ; nlink = 0 ; linbank = 0
```

Let us explain the above shown result. The “neq” is the number of MRHS Linear Equations we have up to that point. The “linbank” (linear bank) is the notation, which shows the number of the ordinary linear equations that have extracted after the application of the agreeing or gluing procedures.

The above system has at the beginning 24 variables combined in 12 MRHS linear equations. Before the application of the agreeing procedure, we cannot extract any ordinary linear equation. That is why we see in the above result that we have 0 number of linear equation in our linear bank (linbank).

This system of equations was used as input in the code *“mrhs,”* which is shown in Appendix B. After the agreeing procedure, we conclude with zero MRHS linear equations and 22 new ordinary linear equations, which appear below.

```
                    0 +   8 +  20 +  21 +  23 = 0
  0 +   1 +   2 +   8 +   9 +  10 +  21 +  22 = 0
                                        21 = 0
                           2 +  10 +  20 = 0
                    1 +   2 +   9 +  10 +  19 = 0
                           0 +   8 +  18 = 0
```

```
                  0 +   2 +   8 +  10 +  17 = 0
                              0 +   8 +  16 = 0
                                         15 = 0
          0 +   1 +   2 +   8 +   9 +  10 +  14 = 0
                  0 +   2 +   8 +  10 +  13 = 0
                  1 +   2 +   9 +  10 +  12 = 0
                  1 +   2 +   9 +  10 +  11 = 0
                              2 +   8 +  10 = 0
                  0 +   1 +   2 +   8 +   9 = 0
                                         8 = 1
                                    2 +   7 = 0
                                    0 +   6 = 0
                                         5 = 0
                                         4 = 0
                              0 +   2 +   3 = 0
                                         1 = 0
now: neq = 0 ; nlink = 0 ; linbank = 22
```

The numbers that appear in the above system of ordinary linear equations represent the number of the variable in our initial system, i.e., 0.

The matrix form of the above ordinary linear equations is shown below:

```
00110000000000000000001
00100000000000000000010
00001100000000000000100
00001000000000000001000
00000011000000000010000
00000010000000000100000
11000000000000001000000
10000000000000010000000
00001100110100010000000
00001000101000100000000
00000011011101000000000
00000010101010000000000
00011100000100000000000
00101000001000000000000
11000001110000000000000
```

```
10000010100000000000000
11101011000000000000000
00110110000000000000000
11111100000000000000000
01001000000000000000000
11110000000000000000000
01000000000000000000000
01010101010001010000000
```

The last row represents the Right Hand Side, which has only 22 elements, meaning that there are two free variables. The same thing can be easily observed from the reduced matrix of the coefficients of the variables. That particular system cannot have a unique solution. The possible solutions of this system appear below.

```
01000000000000000000000 0
10110000000000000000000 0
00001000000000000000000 0
00000100000000000000000 0
10000010000000000000000 0
00100001000000000000000 0
00000000100000000000000 1
10100000010000000000000 1
00100000001000000000000 1
10100000000100000000000 0
10100000000010000000000 0
10000000000001000000000 0
00100000000000100000000 1
00000000000000010000000 0
10000000000000001000000 1
10000000000000000100000 0
10100000000000000010000 1
00000000000000000001000 1
00000000000000000000100 0
00100000000000000000010 1
```

```
100000000000000000000001 0

free:
    0    2


solutions:
?0??00??1??????0????10??
00000000111000101010101O
10010010101111100101101l
00110001100110001011100O
10100011110001000100100l
```

Therefore, the variables x_0 and x_2 are free variables, as it is appeared in the code above. Additionally, we have a depiction of the four different solutions at the end, which means that this particular system has four different keys for one plaintext and ciphertext. In that example, where we have a small-scale variant of the AES with two rounds, two rows and two columns in the rectangular arrangement of the input and two-bit words, the algorithm cannot give a unique solution.

This is a surprising result, as we mentioned in the introduction. We have a specific plaintext, which is encrypted by using a specific key. In our attempt to break the algorithm and to extract the initial plaintext from the ciphertext, we expect to recover the initial key. Instead, solving the system of the MRHS Linear Equations, we found four different keys that can decrypt that ciphertext to get the correct plaintext.

One thing that one could suppose is that some other linear equations could be extracted in order to solve this problem, by applying a *different* plaintext with the same key. At first glance, it can be seen that the ordinary linear equations are different from the previous ones. However, if the matrix of the coefficients of the variables is reduced, we conclude exactly in the same set of linear equations as before. Therefore, such a system has a non-unique solution.

2. Example 2

The second example is one with a unique solution. The parameters of this system are one round, two rows, and two columns in the rectangular arrangement of the input and four bit words.

The system, which appears below, has 24 variables and six MRHS linear equations. Therefore, in order to end up with a unique solution, it is necessary to find 24 ordinary linear equations.

```
  24  6
 8 16
000001000000000000000000
000100000000000000000000
001000000000000000000000
000110000000000000000000
010000000000000000000000
011111010000000000000000
011011100000000000000000
111011100000000000000000
10000000
01000000
10100000
11110000
00001000
11001100
10101100
01101100
00010110
11000101
10110101
11100110
00001101
01011010
00101101
01101110
 8 16
```

```
00000000010000000000000
00000001000000000000000
00000010000000000000000
00000001100000000000000
00000100000000000000000
00000011000100000000000
00000010001000000000000
00001000000000000000000
10000000
01000000
10100000
11110000
00001000
11001100
10101010
01101010
00010111
11000011
10110011
11100001
00001101
01011011
00101101
01101001
 8 16
00000000000000010000000
00000000000100000000000
00000000001000000000000
00000000000110000000000
00000000100000000000000
00000000001100010000000
00000000010011000000000
00000000100000000000000
10000000
11000000
00100000
```

```
01110000
10001000
11001100
10101010
11101010
00010101
01000011
10110011
01100011
10001101
01011001
00101101
01101011
 8 16
1000000000000000000000000
1001000000000000000000000
1010000000000000000000000
1001000000000000010000000
1100000000000000000000000
0011000000000000000010000
0010000000000000000100000
0110000000000000011100000
10000000
11000000
10100000
11110000
10001000
11001100
10101010
11101010
01101111
00111010
01001010
00011001
01110100
00100011
```

```
01010100
00010001
 8 16
000000000000000000000100
000000010000000000010000
000000100000000000100000
000000010000000000011000
000001000000000001000000
000000110000000000110001
000001100000000001101110
000011100000000011101110
10000000
01000000
10100000
11110000
00001000
11001100
10101000
01101000
00010110
11000001
10110001
11100010
00001101
01011010
00101101
01101010
 8 16
000000000000010000000000
000000000001000000000001
000000000010000000000010
000000000011100000000001
000000001000000000000100
000000000011000100000011
000000000110111000000110
000000001110111000001110
```

```
10000000
01000000
10100000
11110000
00001000
11001100
10101000
01101000
00010110
11000001
10110001
11100010
00001101
01011010
00101101
01101010
now: neq = 6 ; nlink = 0 ; linbank = 0
```

In this case, we tried to extract some ordinary linear equations directly from the initial system of the six MRHS Linear Equations, without any result. After the agreeing algorithm is applied, there is still no linear equation in the linear bank. As we said before, “linbank” (Linear bank) is the name of the place in the computer program where all the ordinary linear equations, that are extracted from the different procedures, are saved. This means that many times the agreeing procedure has no result, if it is applied alone in such a system of MRHS Linear Equations. Here is where the gluing procedure is performed. In this particular example, we observe that exactly 24 different ordinary linear equations are extracted, which are shown below.

```
4 +  23 = 0
4 +  22 = 0
4 +  21 = 0
4 +  20 = 0
```

```
                19 = 0
          17 +  18 = 0
                17 = 0
                16 = 0
           4 +  15 = 0
                14 = 0
           4 +  13 = 0
                12 = 0
           4 +  11 = 0
                10 = 0
           4 +   9 = 0
           4 +   8 = 0
     4 +   6 +   7 = 0
           4 +   6 = 0
                 5 = 0
           3 +   4 = 0
                 3 = 1
                 2 = 0
                 1 = 0
                 0 = 0
now: neq = 0 ; nlink = 0 ; linbank = 24
```

As we mentioned before, we conclude in zero MRHS Linear Equations and 24 ordinary linear equations (“linbank=24”).

In a matrix form, these are:

```
24  0
 linear eqs (24):
 24 1
000010000000000000000001
000010000000000000000010
000010000000000000000100
000010000000000000001000
000000000000000000010000
000000000000000001100000
000000000000000001000000
000000000000000010000000
```

```
000010000000000100000000
000000000000001000000000
000010000000010000000000
000000000000100000000000
000010000001000000000000
000000000010000000000000
000010000100000000000000
000010001000000000000000
000010110000000000000000
000010100000000000000000
000001000000000000000000
000110000000000000000000
000100000000000000000000
001000000000000000000000
010000000000000000000000
100000000000000000000000
000000000000000000001000
```

In addition, the unique solution of this system is:

```
100000000000000000000000 0
010000000000000000000000 0
001000000000000000000000 0
000100000000000000000000 1
000010000000000000000000 1
000001000000000000000000 0
000000100000000000000000 1
000000010000000000000000 0
000000001000000000000000 1
000000000100000000000000 1
000000000010000000000000 0
000000000001000000000000 1
000000000000100000000000 0
000000000000010000000000 1
000000000000001000000000 0
000000000000000100000000 1
000000000000000010000000 0
```

```
000000000000000001000000 0
000000000000000000100000 0
000000000000000000010000 0
000000000000000000001000 1
000000000000000000000100 1
000000000000000000000010 1
000000000000000000000001 1

free:

solutions:
000110101101010100001111
```

There are no free variables. Therefore, it is a unique solution, as it appears above.

From the two examples above, we saw how the algebraic attack method of MRHS Linear Equations works. The many experiments that were tested for this thesis were mostly successful for a small variant of the AES. The parameters of the system that were used were at most four rounds, with four rows and columns and 8-bit words instead of 10 rounds that the original AES algorithm uses.

IV. CONCLUSIONS AND RECOMMENDATIONS

This work has shown that the new method of breaking the Rijndael algorithm with the algebraic representation of a system with Multiple Right Hand Sides linear equations is quite effective. It is comparatively effective because by using other algebraic methods, such as the Buchberger's and the F_4 – F_5 algorithms, it was impossible to find a solution for some very simple systems. In particular, a small version of the AES with 5 rounds and 1 row and 1 column in the rectangular arrangement of the input, could not be solved with the older algebraic methods [5]. Moreover, a system with 4 rounds and 1 row and 1 column in the rectangular arrangement of the input, took 20286.18 seconds to be solved by the standard algebraic methods [5]. The MRHS method solved this problem in only 0.032 seconds [5].

In Chapter II, we saw that the new method seems to be much better than the other algebraic methods in terms of complexity. As mentioned before, due to the different parameters that this algorithm can have, (different number of rounds, rows and columns) it is very complicated to specify an exact complexity. According to the real parameters of the AES, our test cannot be considered very realistic, since the values of the parameters of the original AES are bigger. Nevertheless, by comparing the complexities in terms of the time consumed for a small-scale variant to be solved, we conclude that it is worthwhile to further develop and improve this method.

The AES, as we explained in the beginning, is a standard in order to secure federal information. With MRHS Linear Equations, we concluded that we can break a small-scale variant of this system. We cannot say that the AES is at risk, but we can say for sure that this new method gives new perspectives in the field of cryptanalysis that may put the AES algorithm at risk.

For example, one new detail in this thesis, is the application of our codes in many different experiments with different parameters. In particular, instead of

the 8-bit field that professors Håvard Raddum and Igor Semaev used, we executed some experiments in the 2-bit and 4-bit fields, that is original. From that, we discovered a result very important in terms of cryptanalysis. There are some cases where (a small variant of) the Rijndael algorithm can have multiple solutions. We demonstrated that in the 2-bit field. However, it may be valid though more difficult of course, even in the 8-bit field, which might reduce the effectiveness of the algorithm. This means that we may have to be cautious when we choose the encryption keys, because multiple decryption keys would make it easier to break the algorithm and further to decrypt and reveal important federal information.

In conclusion, our first goal was to create the code, which could solve a small-scale variant of the AES. Based on the codes of Professors Raddum and Semaev, we reached this point. In a few cases, a unique solution of a small-scale variant of the AES resulted. However, the small AES algorithm has some weaknesses, which consist of the fact that some systems cannot uniquely be solved, even if different plaintexts with the same initial key are applied.

The complexity of this algorithm is the area that needs to be improved in future research. The reduction of the complexity by a significant factor could upgrade the MRHS algorithm to a very effective algebraic attack method against new very strong cryptographic algorithms. This reduction could be achieved by reducing either the number of the agreeings between the symbols or the numbers of gluings which adds more complexity in the whole algorithm. One way to achieve that could be the simultaneous agreeing of more than two symbols, which may require a smaller number of necessary gluings for the extraction of the ordinary linear equations. In that way, we could break an AES algorithm with parameters of greater value.

APPENDIX A

(These codes are only intended for experiments with the MRHS method, and neither their author nor the author of this thesis takes any responsibility for their use)

1. BASIS.h (by Havard Raddum and Igor Semaev)

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

typedef unsigned int u32;
typedef unsigned char u8;

#define MAXINT 2147483647
#define EPSILON 0.001
#define log2(x) ( 1.44269504088896340735992468l * log(x) )  // !!
changed for bcc - DC

int NVAR, NWORDS;

u8
weight[256]={0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,2,2,3,2,3,3,4,2,3,3,4,3,
4,4,5,1,2,2,3,2,3,3,4,

      2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,1,2,2,3,2,3,3,4,2
,3,3,4,3,4,4,5,

      2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3
,4,4,5,4,5,5,6,

      4,5,5,6,5,6,6,7,1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3
,4,4,5,4,5,5,6,

      2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,2
,3,3,4,3,4,4,5,

      3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,3,4,4,5,4,5,5,6,4
,5,5,6,5,6,6,7,
            4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8};

struct bitVector{
  u32 *v;
  int length, wl;
};

struct eqSymbol{
  int nlin, nrhs, nw, nn, eqnr, originalNRHS;
  u32 **A, **b, **coverID;
  u8 *RHSexists, delSinceExtract;
  struct linkSymbol **link;
```

```
};

struct linkSymbol{
  int ncells, *nCover[2], linknr;
  struct eqSymbol *nlist[2];
  struct bitVector *cellID;
  u8 *cellExists;
};

struct system{
  int neq, nlink;
  struct eqSymbol *E, *linbank;
  struct linkSymbol *L;
};

int ww(u32 *word, int n){
  int i, j, w=0;

  for(i=0; i<n; ++i){
    w+=weight[word[i]&0xff];
    w+=weight[(word[i]>>8)&0xff];
    w+=weight[(word[i]>>16)&0xff];
    w+=weight[word[i]>>24];
  }

  return w;
}

double averageNRHS(struct system *S){
  int i, totnrhs=0;

  if(S->neq==0)
    return 0.0;
  for(i=0; i<S->neq; ++i)
    totnrhs+=S->E[i].nrhs;

  return (double)(totnrhs)/(double)(S->neq);
}

int smallestSetBit(u32 *M, int ncw){
  int j=0, m=0;
  u32 w;

  while(j<ncw && !M[j]){
    j++;
    m+=32;
  }
  if(j==ncw)
    return -1;
  w=M[j];
  for(j=4; j>=0; --j){
    if(!(w&((1<<(1<<j))-1))){
      m+=(1<<j);
      w>>=(1<<j);
    }
```

```
  }

  return m;
}

int largestSetBit(u32 *M, int ncw){
  int j, m;
  u32 w;

  j=ncw-1;
  m=(ncw<<5)-1;
  while(j>=0 && !M[j]){
    j--;
    m-=32;
  }
  if(j==-1)
    return -1;
  w=M[j];
  for(j=4; j>=0; --j){
    if(!(w&(((1<<(1<<j))-1)^0xffffffff)))//no set bit in upper remaning
half
      m-=(1<<j);
    else
      w>>=(1<<j);
  }
  return m;
}
u8 v0GreaterThanV1(struct bitVector v0, struct bitVector v1){
  int i;

  if(v0.length!=v1.length)printf("(v0GreaterThanV1)Uncomparable bit-
strings, v0.length=%d, v1.length=%d!\n",
                          v0.length,v1.length);
  i=v0.wl-1;
  while(i>=0 && !(v0.v[i]^v1.v[i]))
    i--;
  if(i>=0 && v0.v[i]>v1.v[i])
    return 1;
  else
    return 0;
}

u8 v0EqualV1(struct bitVector v0, struct bitVector v1){
  int i;

  if(v0.length!=v1.length)printf("(v0EqualV1)Uncomparable bit-strings,
v0.length=%d, v1.length=%d!\n",
                          v0.length,v1.length);
  i=v0.wl-1;
  while(i>=0 && !(v0.v[i]^v1.v[i]))
    i--;
  if(i<0)
    return 1;
  else
    return 0;
```

```
}

void mergeSortBitVectors(struct bitVector *vl, int nil){
  int i, j, t, na, nb;
  struct bitVector tmp, *ml, *vlb;

  if(nil==1)
    return;
  if(nil==2){
    if(v0GreaterThanV1(vl[0],vl[1])){
      tmp=vl[0];
      vl[0]=vl[1];
      vl[1]=tmp;
    }
    return;
  }
  na=nil/2;
  nb=nil-na;
  vlb=vl+na;
  mergeSortBitVectors(vl,na);
  mergeSortBitVectors(vlb,nb);
  ml=(struct bitVector *)malloc(nil*sizeof(struct bitVector));
  t=i=j=0;
  while(i<na && j<nb){
    if(v0GreaterThanV1(vl[i],vlb[j]))
      ml[t++]=vlb[j++];
    else
      ml[t++]=vl[i++];
  }//lists merged
  if(i<na){
    for(j=i; j<na; ++j)
      ml[t++]=vl[j];
  }
  else{
    for(i=j; i<nb; ++i)
      ml[t++]=vlb[i];
  }//remainder of unfinished list copied
  for(i=0; i<nil; ++i)
    vl[i]=ml[i];
  //copied back to vl
}

void printLinComb(u32 *l, int nvar){
  int i;

  for(i=0; i<nvar; ++i){
    if(l[i>>5]&(1<<(i&0x1f)))
      printf("%3d + ",i);
  }
  printf("\b\b\n");
}

void printEquation(struct eqSymbol *eq){
  /* Prints the equation to the screen, includes up to 16 righ-hand
sides */
```

```
  int i, j, maxnv=0, nv, nr, nwritten;

  if(eq->nrhs>16)
    nr=16;
  else
    nr=eq->nrhs;
  for(i=0; i<eq->nlin; ++i){
    nv=ww(eq->A[i],NWORDS);
    if(nv>maxnv)
      maxnv=nv;
  }
  printf("========= Equation %d ================================\n",eq-
>eqnr);
  for(i=0; i<eq->nlin; ++i){
    nv=ww(eq->A[i],NWORDS);
    for(j=0; j<maxnv-nv; ++j)
      printf("      ");
    for(j=0; j<NVAR; ++j){
      if(eq->A[i][j>>5]&(1<<(j&0x1f)))
      printf(" %3d +",j);
    }
    printf("\b=");
    nwritten=0;
    for(j=0; j<eq->originalNRHS; ++j){
      if(eq->RHSexists[j]){
      if(eq->b[j][i>>5]&(1<<(i&0x1f)))
        printf(" 1 |");
      else
        printf(" 0 |");
      nwritten++;
      if(nwritten==nr)
        j=eq->originalNRHS;
      }
    }
    printf("\b\n\n");
  }
  printf("%d right hand sides in total\n",eq->nrhs);
  printf("\nLinked to %d other equations\n",eq->nn);
  printf("RHS index - ");
  nwritten=0;
  for(i=0; i<eq->originalNRHS; ++i){
    if(eq->RHSexists[i]){
      printf(" %2d ",i);
      nwritten++;
      if(nwritten==nr)
      i=eq->originalNRHS;
    }
  }
  printf("\ncoverID");
  for(i=0; i<eq->nn; ++i){
    printf("\nLink %2d     ",i);
    nwritten=0;
    for(j=0; j<eq->originalNRHS; ++j){
      if(eq->RHSexists[j]){
      printf(" %2d ",eq->coverID[i][j]);
```

```
      nwritten++;
      if(nwritten==nr)
        j=eq->originalNRHS;
      }
    }
  }
  printf("\n=======================================================\n");
}

void printLink(struct linkSymbol *ls){
  /* Prints the link to the screen */
  int i, c0, c1;

  printf("============== Link %d ======================\n",ls-
>linknr);
  printf("Link with %d cells\n",ls->ncells);
  if(ls->ncells<=32){
    printf("\n       ");
    for(i=0; i<ls->ncells; ++i)
      printf(" %2d ",i);
    printf("\nExists ");
    for(i=0; i<ls->ncells; ++i){
      if(ls->cellExists[i])
      printf("  x ");
      else
      printf("    ");
    }
    printf("\nnCover0");
    for(i=0; i<ls->ncells; ++i)
      printf(" %2d ",ls->nCover[0][i]);
    printf("\nnCover1");
    for(i=0; i<ls->ncells; ++i)
      printf(" %2d ",ls->nCover[1][i]);
  }
  else{
    c0=ls->nCover[0][0];
    c1=ls->nCover[1][0];
    for(i=1; i<ls->ncells; ++i){
      if(c0!=ls->nCover[0][i] || c1!=ls->nCover[1][i]){
      printf("\nUnbalanced link\n");
      i=ls->ncells;
      }
    }
  }
  printf("\n\nLinks together equations %d and %d of dimensions %d and
%d\n",
      ls->nlist[0]->eqnr,ls->nlist[1]->eqnr,ls->nlist[0]->nlin,ls-
>nlist[1]->nlin);
  printf("=======================================================\n");
}

void printLinEquation(struct eqSymbol *eq, int start, int stopp){
  int i, j, nv, maxnv=0; // !! added init , maxnv;

  for(i=start; i<stopp; ++i){
```

```
    nv=ww(eq->A[i],NWORDS);
    if(nv>maxnv)
      maxnv=nv;
  }
  printf("========= Linear Equation ====================\n",eq->eqnr);
  for(i=start; i<stopp; ++i){
    nv=ww(eq->A[i],NWORDS);
    for(j=0; j<maxnv-nv; ++j)
      printf("      ");
    for(j=0; j<NVAR; ++j){
      if(eq->A[i][j>>5]&(1<<(j&0x1f)))
      printf(" %3d +",j);
    }
    printf("\b=");
    if(eq->b[0][i>>5]&(1<<(i&0x1f)))
      printf(" 1\n");
    else
      printf(" 0\n");
  }
}

void deleteEquation(struct eqSymbol *eq){
  /* Frees all allocated memory in eq not associated to links. */
  int i;

  for(i=0; i<eq->originalNRHS; ++i){
    if(eq->RHSexists[i])
      free(eq->b[i]);
  }
  free(eq->b);
  for(i=0; i<eq->nlin; ++i)
    free(eq->A[i]);
  free(eq->A);
  free(eq->RHSexists);
}

void deleteSystem(struct system *S){
  int i;

  for(i=0; i<S->neq; ++i)
    deleteEquation(S->E+i);
  deleteEquation(S->linbank);
}

void Uxb(u32 **U, int nr, int nc, u32 *b, u32 *x){
  /* computes U times b, where U has nr rows and nc columns.  Stores
     result in x */
  int i, j, ncw, nrw;
  u32 *w;

  ncw=(nc+31)>>5;
  nrw=(nr+31)>>5;
  w=(u32 *)malloc(ncw*sizeof(u32));
  for(i=0; i<nrw; ++i)
    x[i]=0;
```

```
  for(i=0; i<nr; ++i){
    for(j=0; j<ncw; ++j)
      w[j]=U[i][j]&b[j];
    if(ww(w,ncw)&1)
      x[i>>5]|=(1<<(i&0x1f));
  }
  free(w);
}

void UAX(u32 **U, u32 **A, u32 **nyA, int nl, int ncw){
  /* Multiplies the matrices U and A, stores result in nyA */
  int i, j, k, *bpos, hw;

  bpos=(int *)malloc(nl*sizeof(int));
  for(i=0; i<nl; ++i){
    hw=0;
    for(j=0; j<nl; ++j){
      if(U[i][j>>5]&(1<<(j&0x1f)))
      bpos[hw++]=j;
    }//found positions where U[i] has 1-bits
    for(j=0; j<ncw; ++j){
      for(k=0; k<hw; ++k)
      nyA[i][j]^=A[bpos[k]][j];
    }
  }
}

int rank(u32 **M, int nr, int nc){
  /* Returns the rank of M */
  int rank, i, j, k, sb, minsb, r, ncw;
  u32 **cM, *tmp;

  ncw=(nc+31)>>5;
  cM=(u32 **)malloc(nr*sizeof(u32));
  for(i=0; i<nr; ++i){
    cM[i]=(u32 *)malloc(ncw*sizeof(u32));
    for(j=0; j<ncw; ++j)
      cM[i][j]=M[i][j];
  }
  for(i=0; i<nr; ++i){
    minsb=nc;
    for(j=i; j<nr; ++j){
      sb=smallestSetBit(cM[j],ncw);
      if(sb!=-1 && sb<minsb){
      r=j;
      minsb=sb;
      }
    }
    if(minsb==nc){
      for(j=0; j<nr; ++j)
      free(cM[j]);
      free(cM);
      return i;
    }
    if(r>i){
```

```
      tmp=cM[i];
      cM[i]=cM[r];
      cM[r]=tmp;
    }
    for(j=i+1; j<nr; ++j){
      if(cM[j][minsb>>5]&(1<<(minsb&0x1f))){
      for(k=0; k<ncw; ++k)
        cM[j][k]^=cM[i][k];
      }
    }
  }
  for(i=0; i<nr; ++i)
    free(cM[i]);
  free(cM);
  return nr;
}

int computeU(struct eqSymbol *eq0, struct eqSymbol *eq1, u32 **U){
  /* Computes U such that UM is triangularized, where M is the
   concatenation of the matrices in eq0 and eq1.  Returns the
   dimension of the shared subspaces of eq0 and eq1. */
  int i, j, k, b, minb, r, enord, uw, rank, sumnl;
  u32 *tmp, enmask, **M;

  sumnl=eq0->nlin+eq1->nlin;
  M=(u32 **)malloc(sumnl*sizeof(u32 *));
  for(i=0; i<sumnl; ++i){
    M[i]=(u32 *)malloc(NWORDS*sizeof(u32));
    if(i<eq0->nlin){
      for(j=0; j<NWORDS; ++j)
      M[i][j]=eq0->A[i][j];
    }
    else{
      for(j=0; j<NWORDS; ++j)
      M[i][j]=eq1->A[i-eq0->nlin][j];
    }
  }
  rank=sumnl;
  uw=(sumnl+31)>>5;
  for(i=0; i<sumnl; ++i){
    U[i]=(u32 *)calloc(uw,sizeof(u32));
    U[i][i>>5]|=(1<<(i&0x1f));
  }
  //U is identity-matrix
  for(i=0; i<sumnl; ++i){
    minb=MAXINT;
    for(j=i; j<sumnl; ++j){
      b=smallestSetBit(M[j],NWORDS);
      if(b!=-1 && b<minb){
      r=j;
      minb=b;
      if(minb==i)//no need to search further
        j=sumnl;
      }
    }
```

```
    if(minb==MAXINT){//only all-zero rows remaining
      rank=i;
      i=sumnl;
    }
    else{
      if(r>i){//need to swap rows
      tmp=M[i];
      M[i]=M[r];
      M[r]=tmp;
      tmp=U[i];
      U[i]=U[r];
      U[r]=tmp;
      }
      enmask=1<<(minb&0x1f);
      enord=minb>>5;
      for(j=i+1; j<sumnl; ++j){//making 0's under leading 1
      if(M[j][enord]&enmask){
        for(k=0; k<NWORDS; ++k)
          M[j][k]^=M[i][k];
        for(k=0; k<uw; ++k)
          U[j][k]^=U[i][k];
      }
      }
    }
  }
  for(i=0; i<sumnl; ++i)
    free(M[i]);
  free(M);

  return sumnl-rank;
}

u8 checkSolution(struct system *S, u8 *val){
  /* Returns 1 if val is a solution to S, 0 if not.
     Prints out the equation number of equations not satisfied. val[i]
is assigned to variable i.*/
  int i, j;
  u32 *propRHS;
  struct eqSymbol *eq;
  u8 bit, funnet, rv=1;


  propRHS=(u32 *)calloc(NWORDS,sizeof(u32));
  for(eq=S->E; eq<S->E+S->neq; ++eq){
    for(i=0; i<eq->nlin; ++i){
      bit=0;
      for(j=0; j<NVAR; ++j){
      if(eq->A[i][j>>5]&(1<<(j&0x1f)) && val[j])
        bit^=1;
      }
      if(bit)
      propRHS[i>>5]|=1<<(i&0x1f);
    }//propRHS maa vaere blant eq->b
    for(i=0; i<eq->originalNRHS; ++i){
      if(eq->RHSexists[i]){
```

```
      funnet=1;
      for(j=0; j<eq->nw; ++j){
        if(propRHS[j]^eq->b[i][j])
          funnet=0;
      }
      if(funnet)
        i=eq->originalNRHS+1;
      }
    }
    if(i==eq->originalNRHS){//eq ikke tilfredsstilt
      printf("Equation %d not satisfied, nlin=%d, nrhs=%d\n",eq-
>eqnr,eq->nlin,eq->nrhs);
      //printEquation(eq);
      rv=0;
    }
    for(i=0; i<eq->nw; ++i)
      propRHS[i]=0;
  }
  return rv;
}

u8 deleteRHS(struct eqSymbol *eq, int RHSindex){
  int i, minindex, ci;
  struct linkSymbol *lS;

  if(eq->RHSexists[RHSindex]==1){
    eq->RHSexists[RHSindex]=0;
    free(eq->b[RHSindex]);
    eq->nrhs--;
    if(eq->nrhs==0)
      return 0;
    for(i=0; i<eq->nn; ++i){
      lS=eq->link[i];
      ci=eq->coverID[i][RHSindex];
      if(lS->nlist[0]==eq)
      minindex=0;
      else
      minindex=1;
      lS->nCover[minindex][ci]--;
      if(lS->nCover[minindex][ci]<0){printLink(lS);exit(0);}
    }
    eq->delSinceExtract=1;
  }
  return 1;
}

u8 trimEquation(struct eqSymbol *eq){
  /* Makes sure that A-matrix in eq has full rank, and removes
     RHS if possible.  Returns 0 if eq can not be satisfied, returns 1
     if A-matrix had full rank, returns 2 if A-matrix has decreased. */
  u32 **U, *x, *tmp, enmask;
  int i, j, k, b, minb, r, enord, uw, rank;
  u8 rv; // !! removed var: , delocc=0;

  rank=eq->nlin;
```

```
  uw=(eq->nlin+31)>>5;
  U=(u32 **)malloc(eq->nlin*sizeof(u32 *));
  for(i=0; i<eq->nlin; ++i){
    U[i]=(u32 *)calloc(uw,sizeof(u32));
    U[i][i>>5]|=(1<<(i&0x1f));
  }
  //U is identity-matrix
  for(i=0; i<eq->nlin; ++i){
    minb=NVAR;
    for(j=i; j<eq->nlin; ++j){
      b=smallestSetBit(eq->A[j],NWORDS);
      if(b!=-1 && b<minb){
      r=j;
      minb=b;
      }
    }
    if(minb==NVAR){//only all-zero rows remaining
      rank=i;
      i=eq->nlin;
    }
    else{
      if(r>i){//need to swap rows
      tmp=eq->A[i];
      eq->A[i]=eq->A[r];
      eq->A[r]=tmp;
      tmp=U[i];
      U[i]=U[r];
      U[r]=tmp;
      }
      enmask=1<<(minb&0x1f);
      enord=minb>>5;
      for(j=i+1; j<eq->nlin; ++j){//making 0's under leading 1
      if(eq->A[j][enord]&enmask){
        for(k=0; k<NWORDS; ++k)
          eq->A[j][k]^=eq->A[i][k];
        for(k=0; k<uw; ++k)
          U[j][k]^=U[i][k];
      }
      }
    }
  }//U is computed
  x=(u32 *)malloc(eq->nw*sizeof(u32));
  for(i=0; i<eq->originalNRHS; ++i){
    if(eq->RHSexists[i]){
      Uxb(U,eq->nlin,eq->nlin,eq->b[i],x);//x er ny RHS-vektor
      for(j=0; j<eq->nw; ++j)
      eq->b[i][j]=x[j];
      for(j=rank; j<eq->nlin; ++j){
      if(eq->b[i][j>>5]&(1<<(j&0x1f))){//inneholder 1-bit i 0-rad
omraade
        rv=deleteRHS(eq,i);
        // !! removed var: delocc=1;
        if(rv==0)
          return 0;
        j=eq->nlin;
```

```
      }
      }
    }
  }//oppdatert alle RHS, og sikret at de er gyldige
  if(eq->nlin==rank)
    return 1;
  else{
    eq->nlin=rank;
    eq->nw=(eq->nlin+31)>>5;
    return 2;
  }
}

void depositLinComb(struct eqSymbol *eq, u32 *lc, u8 rhv){
  /* Adds lc=rhv to eq (the linear bank).  The bank must be
     antitriangular, and will remain so after addition of lc=rhv.
     lc must not be in the span of eq->A. */
  int i, j, vnr, insertrow=-1, libit;
  u32 *clc;

  clc=(u32 *)malloc(NWORDS*sizeof(u32));
  for(i=0; i<NWORDS; ++i)
    clc[i]=lc[i];
  //clc er kopi og kan herjes med
  if(eq->nlin==0)
    insertrow=0;
  i=0;
  while(insertrow==-1){
    vnr=largestSetBit(clc,NWORDS);
    while(i<eq->nlin && vnr<largestSetBit(eq->A[i],NWORDS))
      i++;
    if(i<eq->nlin){
      libit=largestSetBit(eq->A[i],NWORDS);
      if(vnr>libit)
      insertrow=i;
      else{//vnr==libit
      for(j=0; j<NWORDS; ++j)
        clc[j]^=eq->A[i][j];
      if(eq->b[0][i>>5]&(1<<(i&0x1f)))
        rhv^=1;
      i++;
      }
    }
    else//i==eq->nlin
      insertrow=eq->nlin;
  }
  for(i=eq->nlin; i>insertrow; --i){
    eq->A[i]=eq->A[i-1];
    if(((eq->b[0][i>>5]>>(i&0x1f))^(eq->b[0][(i-1)>>5]>>((i-
1)&0x1f)))&1)
      eq->b[0][i>>5]^=(1<<(i&0x1f));
  }//ryddet plass til ny linear ligning
  eq->A[insertrow]=clc;
  if((u8)((eq->b[0][insertrow>>5]>>(insertrow&0x1f))&1)^rhv)//feil bit
i RHS
```

```
    eq->b[0][insertrow>>5]^=(1<<(insertrow&0x1f));
  eq->nlin++;
}

u8 substituteLinComb(struct system *S, u32 *lc, u32 rhv){
  /* Largest variable in lc is eliminated using the equation lc = rhv.
     Returns 0 if system becomes inconsistent, 1 if no A-matrices were
     reduced, and 2 if some A-matrices did not have full rank after
     substitution. */
  int i, j, enord, maxvar, rhsshift, rhsord, sistshift, sisteord;
  u32 enmask;
  struct eqSymbol *eq;
  u8 eqendret, lokrv, rv=1;

  maxvar=largestSetBit(lc,NWORDS);
  enmask=(1<<(maxvar&0x1f));
  enord=maxvar>>5;
  for(eq=S->E; eq<S->E+S->neq; ++eq){
    eqendret=0;
    for(i=0; i<eq->nlin; ++i){
      if(eq->A[i][enord]&enmask){//rekke i i eq har variabelen som skal
erstattes
      eqendret=1;
      for(j=0; j<NWORDS; ++j)
        eq->A[i][j]^=lc[j];
      if(rhv){//maa endre alle rhs
        rhsshift=1<<(i&0x1f);
        rhsord=i>>5;
        for(j=0; j<eq->originalNRHS; ++j){
          if(eq->RHSexists[j])
            eq->b[j][rhsord]^=rhsshift;
        }
      }
      }
    }
    if(eqendret){
      lokrv=trimEquation(eq);
      if(lokrv==0){
      printf("trimeq returnerer 0 for ligning %d\n",eq->eqnr);
      return 0;
      }
      if(lokrv==2)
      rv=2;
    }
  }
  depositLinComb(S->linbank,lc,rhv);
  //substituted equation added to the bank

  return rv;
}

void trimSystem(struct system *S){
  /* Removes equations with no information in them. */
  struct eqSymbol *eq;
```

```
  for(eq=S->E; eq<S->E+S->neq; ++eq){
    if(eq->nlin<32 && (1<<eq->nlin)==eq->nrhs){//ligning uten
informasjon
      if(eq<S->E+S->neq-1)
      *eq=S->E[S->neq-1];
      eq--;
      S->neq--;
    }
  }
}

void initLinBank(struct eqSymbol *leq){
  leq->A=(u32 **)malloc(2*NVAR*sizeof(u32 *));
  leq->b=(u32 **)malloc(sizeof(u32 *));
  leq->b[0]=(u32 *)calloc(2*NWORDS,sizeof(u32));
  leq->nlin=0;
  leq->nrhs=1;
  leq->nw=0;
  leq->nn=0;
  leq->eqnr=-1;
  leq->originalNRHS=1;
  leq->RHSexists=(u8 *)malloc(sizeof(u8));
  leq->RHSexists[0]=1;
}

u8 solved(struct system *S){
  int i;

  if(S->neq==0)
    return 1;
  if(S->neq==1 && S->E[0].nrhs>=1)
    return 1;
  for(i=0; i<S->neq; ++i){
    if(S->E[i].nrhs!=1)
      return 0;
  }
  return 1;
}

void solveLinSystem(struct eqSymbol *e){
  int i, j, k, sb, minb, maxb, r;
  u32 *tmp;
  u8 ibit, rbit;

  for(i=0; i<e->nlin; ++i){
    minb=NVAR;
    for(j=i; j<e->nlin; ++j){
      sb=smallestSetBit(e->A[j],NWORDS);
      if(sb>=0 && sb<minb){
      minb=sb;
      r=j;
      }
      if(minb==i)
      j=e->nlin;
    }
```

```
    if(e->b[0][i>>5]&(1<<(i&0x1f)))
      ibit=1;
    else
      ibit=0;
    if(r!=i){//trenger å bytte om på rader
      tmp=e->A[i];
      e->A[i]=e->A[r];
      e->A[r]=tmp;
      if(e->b[0][r>>5]&(1<<(r&0x1f)))
      rbit=1;
      else
      rbit=0;
      if(ibit^rbit){
      e->b[0][i>>5]^=(1<<(i&0x1f));
      ibit^=1;
      e->b[0][r>>5]^=(1<<(r&0x1f));
      }
    }
    for(j=i+1; j<e->nlin; ++j){//lager 0 under ledende 1'er
      if(e->A[j][minb>>5]&(1<<(minb&0x1f))){
      for(k=minb>>5; k<NWORDS; ++k)
        e->A[j][k]^=e->A[i][k];
      if(ibit)
        e->b[0][j>>5]^=(1<<(j&(0x1f)));
      }
    }
  }//triangularisert, starter tilbakesubstitusjon

  for(i=e->nlin-1; i>0; --i){
    maxb=largestSetBit(e->A[i],NWORDS);
    if(e->b[0][i>>5]&(1<<(i&0x1f)))
      ibit=1;
    else
      ibit=0;
    for(j=i-1; j>=0; --j){
      if(e->A[j][maxb>>5]&(1<<(maxb&0x1f))){
      for(k=0; k<NWORDS; ++k)
        e->A[j][k]^=e->A[i][k];
      if(ibit)
        e->b[0][j>>5]^=(1<<(j&0x1f));
      }
    }
  }
}

void copySystem(struct system *S, struct system *kopiS){
  int i, j, k;

  kopiS->neq=S->neq;
  kopiS->E=(struct eqSymbol *)malloc(kopiS->neq*sizeof(struct
eqSymbol));
  kopiS->linbank=(struct eqSymbol *)malloc(sizeof(struct eqSymbol));
  initLinBank(kopiS->linbank);
  for(i=0; i<S->linbank->nlin; ++i){
    for(j=0; j<NWORDS; ++j)
```

```
      kopiS->linbank->A[i][j]=S->linbank->A[i][j];
  }
  for(i=0; i<S->linbank->nw; ++i)
    kopiS->linbank->b[0][i]=S->linbank->b[0][i];
  kopiS->linbank->nlin=S->linbank->nlin;
  kopiS->linbank->nw=S->linbank->nw;
  for(i=0; i<S->neq; ++i){
    kopiS->E[i].nlin=S->E[i].nlin;
    kopiS->E[i].nrhs=S->E[i].nrhs;
    kopiS->E[i].originalNRHS=S->E[i].originalNRHS;
    kopiS->E[i].nw=S->E[i].nw;
    kopiS->E[i].nn=0;
    kopiS->E[i].A=(u32 **)malloc(kopiS->E[i].nlin*sizeof(u32 *));
    for(j=0; j<kopiS->E[i].nlin; ++j){
      kopiS->E[i].A[j]=(u32 *)malloc(NWORDS*sizeof(u32));
      for(k=0; k<NWORDS; ++k)
      kopiS->E[i].A[j][k]=S->E[i].A[j][k];
    }
    kopiS->E[i].b=(u32 **)malloc(kopiS->E[i].nrhs*sizeof(u32 *));
    for(j=0; j<kopiS->E[i].nrhs; ++j){
      kopiS->E[i].b[j]=(u32 *)malloc(kopiS->E[i].nw*sizeof(u32));
      for(k=0; k<kopiS->E[i].nw; ++k)
      kopiS->E[i].b[j][k]=S->E[i].b[j][k];
    }
    kopiS->E[i].RHSexists=(u8 *)malloc(kopiS-
>E[i].originalNRHS*sizeof(u8));
    for(j=0; j<kopiS->E[i].originalNRHS; ++j)
      kopiS->E[i].RHSexists[j]=S->E[i].RHSexists[j];
    kopiS->E[i].delSinceExtract=S->E[i].delSinceExtract;
  }
}

double log2sum(double *T, int n){
  /* computes log2(\sum_{i=0}^{n-1}(2^{T[i]})).  Works also when the
sum is greater than 2^{32}. */
  double suma, sumb;
  int na;

  if(n==1)
    return T[0];
  na=n/2;
  suma=log2sum(T,na);
  sumb=log2sum(T+na,n-na);
  if(suma<sumb){
    if((sumb-28.0)>suma)//only sumb contributes
      return sumb;
    else//compute exactly
      return suma+log2(pow(2,sumb-suma)+1.0);
  }
  else{//suma largest
    if((suma-28.0)>sumb)//only suma contributes
      return suma;
    else//compute exactly
      return sumb+log2(pow(2,suma-sumb)+1.0);
  }
```

```
}

u8 evaluate(u32 *lc, u8 *v){
  /* evaluates the linear combination lc with the variables in v and
returns its sum */
  int i;
  u8 sum=0;

  for(i=0; i<NVAR; ++i){
    if(lc[i>>5]&(1<<(i&0x1f)) && v[i])
      sum^=1;
  }

  return sum;
}
```

2. AGREEING.h (by Havard Raddum and Igor Semaev)

```
u8 checkLink(struct linkSymbol *lS){
  /*Checks if the two equations linked in this link agree.  Returns 0
if equations are inconsistent,
    1 if equations agree and 2 if equations disagreed */
  u8 rv=1, lokrv;
  int i, j, eqdel, linkIndex, eq0ni, eq1ni;
  struct eqSymbol *eq, *eq0, *eq1;

  if((log2(lS->ncells)<(log2(lS->nlist[0]->nrhs)/2)) &&
     (log2(lS->ncells)<(log2(lS->nlist[1]->nrhs)/2))){//do not expect
deletions
    for(i=0; i<lS->ncells; ++i){
      if(lS->cellExists[i]){
      eqdel=2;
      if(lS->nCover[0][i]==0 && lS->nCover[1][i]>0)
        eqdel=1;
      if(lS->nCover[0][i]>0 && lS->nCover[1][i]==0)
        eqdel=0;
      if(eqdel<2){
        rv=2;
        eq=lS->nlist[eqdel];
        for(j=0; j<eq->nn; ++j){
          if(eq->link[j]==lS){
            linkIndex=j;
            j=eq->nn+1;
          }
        }
        if(j==eq->nn){
          printf("Fant ikke link!?  eqdel=%d, i=%d\n",eqdel,i);
          printf("Linkadresse: %x, eq.naboadresse: %x\n",lS,eq-
>link[0]);
        }

        j=0;
```

```
        while(lS->nCover[eqdel][i]>0){
          while(j<eq->originalNRHS && eq-
>coverID[linkIndex][j]!=(unsigned)i) // !! added cast
            ++j;
          if(j==eq->originalNRHS)printf("fant ikke RHS som dekker celle
%x, lS->nCover=%d\n",i,lS->nCover[eqdel][i]);
          lokrv=deleteRHS(eq,j);
          if(lokrv==0)
            return 0;
          ++j;
        }
        lS->cellExists[i]=0;
      }
      }
    }
  }
  else{//expects deletions, goes through RHS's of linked equations
    eq0=lS->nlist[0];
    eq1=lS->nlist[1];
    for(i=0; i<eq0->nn; ++i){
      if(eq0->link[i]==lS){
      eq0ni=i;
      i=eq0->nn+1;
      }
    }
    if(i==eq0->nn)printf("(checkLink)Fant ikke link fra nabo 0\n");
    for(i=0; i<eq1->nn; ++i){
      if(eq1->link[i]==lS){
      eq1ni=i;
      i=eq1->nn+1;
      }
    }
    if(i==eq1->nn)printf("(checkLink)Fant ikke link fra nabo 1\n");
    for(i=0; i<eq1->originalNRHS; ++i){
      if(eq1->RHSexists[i]){
      if(lS->nCover[1][eq1->coverID[eq1ni][i]]==0)
        printf("RHS %d i eq1 dekker en celle som link sier ikke blir
dekket av eq1\n",i);
      if(lS->nCover[0][eq1->coverID[eq1ni][i]]==0){
        lokrv=deleteRHS(eq1,i);
        if(lokrv==0)
          return 0;
        rv=2;
      }
      }
    }
    for(i=0; i<eq0->originalNRHS; ++i){
      if(eq0->RHSexists[i]){
      if(lS->nCover[0][eq0->coverID[eq0ni][i]]==0)
        printf("RHS %d i eq0 dekker en celle som link sier ikke blir
dekket av eq0\n",i);
      if(lS->nCover[1][eq0->coverID[eq0ni][i]]==0){
        lokrv=deleteRHS(eq0,i);
        if(lokrv==0)
          return 0;
```

```
        rv=2;
      }
      }
    }
  }
  return rv;
}

u8 agreeSystem(struct system *S){
  /* Agrees the whole system.  Returns 0 if system is inconsistent,
     1 if system already agreed, or 2 if deletions of RHS's have
occurred */
  int i;
  u8 rv=1, lokrv, changed=1;
  struct linkSymbol *lsym;

  while(changed){
    changed=0;
    for(lsym=S->L; lsym<S->L+S->nlink; ++lsym){
      lokrv=checkLink(lsym);
      if(lokrv==2)
      changed=1;
      if(lokrv==0){
      printf("inconsistency in link %d\n",i);
      return 0;
      }
    }
  }

  return rv;
}

      }
```

3. GLUING.h (by Havard Raddum and Igor Semaev)

```
/* Methods inmplementing gluing of equations, before linking */

u8 maskedRHS1EqualRHS2(u32 *rhs1, u32 *rhs2, int nw, u32 *maske){
  int t;

  t=nw-1;
  while(t>=0 && ((rhs1[t]&maske[t])==(rhs2[t]&maske[t])))
    t--;
  if(t<0)
    return 1;
  else
    return 0;
}

u8 maskedRHS1BiggerThanRHS2(u32 *rhs1, u32 *rhs2, int nw, u32 *maske){
  int t;

  t=nw-1;
  while(t>=0 && (rhs1[t]&maske[t])==(rhs2[t]&maske[t]))
```

```
    t--;
  if(t>=0 && (rhs1[t]&maske[t])>(rhs2[t]&maske[t]))
    return 1;
  else
    return 0;
}

void mergeSortMaskedRightHandSides(u32 **RHS, int nicl, int nw, u32
*maske){
  int niacl, nibcl, i, j, t;
  u32 *tmp, **aRHS, **bRHS, **dl;

  if(nicl==2 && maskedRHS1BiggerThanRHS2(RHS[0],RHS[1],nw,maske)){
    tmp=RHS[0];
    RHS[0]=RHS[1];
    RHS[1]=tmp;
  }
  if(nicl>2){
    niacl=nicl/2;
    nibcl=nicl-niacl;
    aRHS=RHS;
    bRHS=RHS+niacl;
    mergeSortMaskedRightHandSides(aRHS,niacl,nw,maske);
    mergeSortMaskedRightHandSides(bRHS,nibcl,nw,maske);
    dl=(u32 **)malloc(nicl*sizeof(u32 *));
    i=j=t=0;
    while(i<niacl && j<nibcl){//ingen er ferdige, må sammenligne
      if(maskedRHS1BiggerThanRHS2(aRHS[i],bRHS[j],nw,maske))
      dl[t++]=bRHS[j++];
      else
      dl[t++]=aRHS[i++];
    }
    if(i==niacl){//aRHS ble ferdig først, kopierer resten av bRHS
      for(i=j; i<nibcl; ++i)
      dl[t++]=bRHS[i];
    }
    else{//bRHS ble ferdig først, kopierer resten av aRHS
      for(j=i; j<niacl; ++j)
      dl[t++]=aRHS[j];
    }
    //kopierer tilbake i RHS
    for(i=0; i<nicl; ++i)
      RHS[i]=dl[i];
    free(dl);
  }
}

void computeMaskedSortedRHS(struct eqSymbol *eq1, struct eqSymbol *eq2,
                      u32 **Ub1, u32 **Ub2, u32 *maske, u32 **U, int
ncommon){
  /* Expands RHS's in eq1 and eq2 so they can be glued together.  maske
     shows which bits that must be equal when gluing.  */
  u32 *lokb;
  int i, j, t, sumnl, sumnlw, skev;
```

```
  sumnl=eq1->nlin+eq2->nlin;
  sumnlw=(sumnl+31)>>5;
  lokb=(u32 *)malloc(sumnlw*sizeof(u32));
  t=0;
  for(i=0; i<eq1->originalNRHS; ++i){//computes Ub for all right-hand
sides b in E1
    if(eq1->RHSexists[i]){
      for(j=0; j<eq1->nw; ++j)
      lokb[j]=eq1->b[i][j];
      for(j=eq1->nw; j<sumnlw; ++j)
      lokb[j]=0;

      Uxb(U,sumnl,sumnl,lokb,Ub1[t++]);
    }
  }
  if(t!=eq1->nrhs)printf("eq1->nrhs=%d, t=%d\n",eq1->nrhs,t);
  t=0;
  for(i=0; i<eq2->originalNRHS; ++i){//computes Ub for all right-hand
sides b in E2
    if(eq2->RHSexists[i]){
      for(j=0; j<eq1->nw; ++j)
      lokb[j]=0;
      skev=(eq1->nlin)&0x1f;
      if(skev==0){
      for(j=eq1->nw; j<sumnlw; ++j)
        lokb[j]=eq2->b[i][j-eq1->nw];
      }
      else{
      for(j=0; j<eq2->nw; ++j){
        lokb[eq1->nw+j-1]|=(eq2->b[i][j]<<skev);
        if(sumnlw>(j+eq1->nw))
          lokb[eq1->nw+j]=eq2->b[i][j]>>(32-skev);
      }
      }
      Uxb(U,sumnl,sumnl,lokb,Ub2[t++]);
    }
  }
  if(t!=eq2->nrhs)printf("eq2->nrhs=%d, t=%d\n",eq2->nrhs,t);

  free(lokb);

  i=sumnl-ncommon;
  j=0;
  while(i>=32){
    maske[j++]=0;
    i-=32;
  }
  if(j<sumnlw){
    maske[j]=(0xffffffff<<i);
    for(i=j+1; i<sumnlw; ++i)
      maske[i]=0xffffffff;
    if(sumnl&0x1f)
      maske[sumnlw-1]&=(1<<(sumnl&0x1f))-1;
  }
  //mask in place
```

```
  mergeSortMaskedRightHandSides(Ub1,eq1->nrhs,sumnlw,maske);
  mergeSortMaskedRightHandSides(Ub2,eq2->nrhs,sumnlw,maske);
}

int nRHSwhenGlued(struct eqSymbol *eq1, struct eqSymbol *eq2){
  /* Computes number of RHS if gluing eq1 and eq2 */
  u32 **Ub1, **Ub2, *maske, **U;
  int i, j, nn=0, bp1, pp1, bp2, pp2, prod, nw, nl, nc;

  nl=eq1->nlin+eq2->nlin;
  nw=(nl+31)>>5;
  U=(u32 **)malloc(nl*sizeof(u32 *));
  nc=computeU(eq1,eq2,U);
  if(nc==0){
    for(i=0; i<nl; ++i)
      free(U[i]);
    free(U);
    if((MAXINT/eq1->nrhs)<eq2->nrhs)
      return MAXINT;
    else
      return eq1->nrhs*eq2->nrhs;
  }
  Ub1=(u32 **)malloc(eq1->nrhs*sizeof(u32 *));
  for(i=0; i<eq1->nrhs; ++i)
    Ub1[i]=(u32 *)calloc(nw,sizeof(u32));
  Ub2=(u32 **)malloc(eq2->nrhs*sizeof(u32 *));
  for(i=0; i<eq2->nrhs; ++i)
    Ub2[i]=(u32 *)calloc(nw,sizeof(u32));
  maske=(u32 *)malloc(nw*sizeof(u32));

  computeMaskedSortedRHS(eq1,eq2,Ub1,Ub2,maske,U,nc);

  for(i=0; i<nl; ++i)
    free(U[i]);
  free(U);
  bp1=pp1=bp2=pp2=0;
  while(bp1<eq1->nrhs && bp2<eq2->nrhs && nn<MAXINT){
    while(bp1<eq1->nrhs && bp2<eq2->nrhs &&
!maskedRHS1EqualRHS2(Ub1[bp1],Ub2[bp2],nw,maske)){
      if(maskedRHS1BiggerThanRHS2(Ub1[bp1],Ub2[bp2],nw,maske))
      bp2++;
      else
      bp1++;
    }//her er maskert Ub1[bp1] og Ub2[bp2] like, eller en av listene er
nådd til endes
    pp1=bp1;
    pp2=bp2;
    while(pp1<eq1->nrhs &&
maskedRHS1EqualRHS2(Ub1[bp1],Ub1[pp1],nw,maske))
      pp1++;
    while(pp2<eq2->nrhs &&
maskedRHS1EqualRHS2(Ub2[bp2],Ub2[pp2],nw,maske))
      pp2++;
    //Blokkene fra bp1 og bp2 til pp1 og pp2 er identiske maskert
    if((pp1-bp1)>0 && MAXINT/(pp1-bp1)<(pp2-bp2))
```

```
      nn=MAXINT;
    prod=(pp1-bp1)*(pp2-bp2);
    if(nn>MAXINT-prod)
      nn=MAXINT;
    if(nn<MAXINT)
      nn+=prod;
    bp1=pp1;
    bp2=pp2;
  }
  for(i=0; i<eq1->nrhs; ++i)
    free(Ub1[i]);
  free(Ub1);
  for(i=0; i<eq2->nrhs; ++i)
    free(Ub2[i]);
  free(Ub2);
  free(maske);

  return nn;
}

int estimateNRHSwhenGlued(struct eqSymbol *eq1, struct eqSymbol *eq2){
  int i, enn, nc, sumnl;
  u32 **M;

  sumnl=eq1->nlin+eq2->nlin;
  M=(u32 **)malloc(sumnl*sizeof(u32 *));
  for(i=0; i<eq1->nlin; ++i)
    M[i]=eq1->A[i];
  for(i=eq1->nlin; i<sumnl; ++i)
    M[i]=eq2->A[i-eq1->nlin];
  nc=sumnl-rank(M,sumnl,NVAR);
  free(M);
  if((log2(eq1->nrhs)+log2(eq2->nrhs)-(double)nc)<30.9){
    if((log2(eq1->nrhs)+log2(eq2->nrhs)-(double)nc)>0.0){
      if(eq2->nrhs>eq1->nrhs)
      enn=(eq1->nrhs>>(nc/2))*(eq2->nrhs>>(nc-(nc/2)));
      else
      enn=(eq2->nrhs>>(nc/2))*(eq1->nrhs>>(nc-(nc/2)));
      if(enn==0)
      enn=1;
    }
    else
      enn=1;
  }
  else
    enn=MAXINT;

  return enn;
}

void glue(struct eqSymbol *eq1, struct eqSymbol *eq2, struct eqSymbol
*geq){
  /* Glues together eq1 and eq2 into geq, eq1 and eq2 should not be
used afterwards. */
  int i, j, k, bp1, pp1, bp2, pp2, ncommon, t, estnnew;
```

```
  u32 *maske, **Ub1, **Ub2, **U, **nyA;

  Ub1=(u32 **)malloc(eq1->nrhs*sizeof(u32 *));
  Ub2=(u32 **)malloc(eq2->nrhs*sizeof(u32 *));
  geq->nlin=eq1->nlin+eq2->nlin;
  geq->nw=(geq->nlin+31)>>5;
  geq->A=(u32 **)malloc(geq->nlin*sizeof(u32 *));
  for(i=0; i<eq1->nlin; ++i)
    geq->A[i]=eq1->A[i];
  for(i=0; i<eq2->nlin; ++i)
    geq->A[eq1->nlin+i]=eq2->A[i];

  nyA=(u32 **)malloc(geq->nlin*sizeof(u32 *));
  U=(u32 **)malloc(geq->nlin*sizeof(u32 *));
  for(i=0; i<geq->nlin; ++i)
    nyA[i]=(u32 *)calloc(NWORDS,sizeof(u32));
  for(i=0; i<eq1->nrhs; ++i)
    Ub1[i]=(u32 *)calloc(geq->nw,sizeof(u32));
  for(i=0; i<eq2->nrhs; ++i)
    Ub2[i]=(u32 *)calloc(geq->nw,sizeof(u32));
  maske=(u32 *)malloc(geq->nw*sizeof(u32));

  ncommon=computeU(eq1,eq2,U);
  UAX(U,geq->A,nyA,geq->nlin,NWORDS);
  free(geq->A);
  geq->A=nyA;
  computeMaskedSortedRHS(eq1,eq2,Ub1,Ub2,maske,U,ncommon);
  estnnew=(eq1->nrhs>>(ncommon/2))*(eq2->nrhs>>(ncommon-ncommon/2));
  if(estnnew<1)
    estnnew=1;
  //printf("estnnew=2^(%.3f), ",log2(estnnew));

  geq->b=(u32 **)malloc(estnnew*sizeof(u32 *));
  bp1=pp1=bp2=pp2=t=0;
  while(bp1<eq1->nrhs && bp2<eq2->nrhs){
    while(bp1<eq1->nrhs && bp2<eq2->nrhs &&
!maskedRHS1EqualRHS2(Ub1[bp1],Ub2[bp2],geq->nw,maske)){
      if(maskedRHS1BiggerThanRHS2(Ub1[bp1],Ub2[bp2],geq->nw,maske))
      bp2++;
      else
      bp1++;
    }//her er maskert Ub1[bp1] og Ub2[bp2] like, eller en av listene er
nådd til endes
    pp1=bp1;
    pp2=bp2;
    while(pp1<eq1->nrhs && maskedRHS1EqualRHS2(Ub1[bp1],Ub1[pp1],geq-
>nw,maske))
      pp1++;
    while(pp2<eq2->nrhs && maskedRHS1EqualRHS2(Ub2[bp2],Ub2[pp2],geq-
>nw,maske))
      pp2++;
    //Blokkene fra bp1 og bp2 til pp1 og pp2 er identiske maskert,
limer alle par
    for(i=bp1; i<pp1; ++i){
      for(j=bp2; j<pp2; ++j){
```

```
      geq->b[t]=(u32 *)malloc(geq->nw*sizeof(u32));
      for(k=0; k<geq->nw; ++k)
        geq->b[t][k]=Ub1[i][k]^Ub2[j][k];//faktisk liming
      t++;
      if(t==estnnew){//needs more memory and a better estimate
        if(i==0)
          estnnew*=2;
        else{
          if(i==eq1->nrhs)
            estnnew+=(pp2-j);
          else
            estnnew=(int)((double)estnnew/((double)i/(double)eq1-
>nrhs));
        }
        geq->b=(u32 **)realloc(geq->b,estnnew*sizeof(u32 *));
      }
      }
    }
    bp1=pp1;
    bp2=pp2;
  }
  geq->originalNRHS=t;
  geq->nrhs=t;
  geq->b=(u32 **)realloc(geq->b,geq->originalNRHS*sizeof(u32 *));
  geq->RHSexists=(u8 *)malloc(geq->originalNRHS*sizeof(u8));
  for(i=0; i<geq->originalNRHS; ++i)
    geq->RHSexists[i]=1;
  if(ncommon>0){//trimmer bort lineært avhengig informasjon
    for(i=geq->nlin-ncommon; i<geq->nlin; ++i)
      free(geq->A[i]);
    geq->nlin-=ncommon;
    geq->A=(u32 **)realloc(geq->A,geq->nlin*sizeof(u32 *));
    geq->nw=(geq->nlin+31)>>5;
    if(((geq->nlin+31)>>5)<geq->nw){
      geq->nw=(geq->nlin+31)>>5;
      for(i=0; i<geq->nrhs; ++i)
      geq->b[i]=(u32 *)realloc(geq->b[i],geq->nw*sizeof(u32));
    }
  }
  geq->nn=0;

  free(maske);
  for(i=0; i<eq1->nrhs; ++i)
    free(Ub1[i]);
  free(Ub1);
  for(i=0; i<eq2->nrhs; ++i)
    free(Ub2[i]);
  free(Ub2);
}

u8 packSystem(struct system *S, int th){
  /* Tries to glue together as many equations as possible, with the
restriction that the number of
     right-hand sides in the glued equation should not be estimated to
be larger than th.
```

```
   Returns 0 if inconsistencies are found, 1 otherwise. */
struct eqSymbol *nyE, *tmp;
int i, j, nnew, minnew, mini, minj, nyneq=0, **gluetab;
u8 *limt, tryglue, trypack;

nyE=(struct eqSymbol *)malloc(S->neq*sizeof(struct eqSymbol));
tmp=(struct eqSymbol *)malloc(sizeof(struct eqSymbol));
limt=(u8 *)calloc(S->neq,sizeof(u8));

gluetab=(int **)malloc(S->neq*sizeof(int *));
for(i=0; i<S->neq; ++i)
  gluetab[i]=(int *)calloc(S->neq,sizeof(int));
for(i=0; i<S->neq; ++i){
  for(j=i+1; j<S->neq; ++j){
    gluetab[i][j]=estimateNRHSwhenGlued(S->E+i,S->E+j);
    if(gluetab[i][j]==0){
    j=S->neq;
    i=S->neq+1;
    }
  }
}
if(i==S->neq)
  tryglue=1;
else{//funnet inkonsistens
  tryglue=2;
  printf("inconsistency found\n");
}
while(tryglue==1){
  minnew=th+1;
  for(i=0; i<S->neq; ++i){
    if(!limt[i]){
    for(j=i+1; j<S->neq; ++j){
      if(!limt[j]){
        nnew=gluetab[i][j];
        if(nnew<minnew){
          minnew=nnew;
          mini=i;
          minj=j;
        }
      }
    }
    }
  }//her er billigste liming funnet, hvis mulig aa lime under th
  if(minnew<=th){//mulig aa lime
    glue(S->E+mini,S->E+minj,nyE+nyneq);
    nyE[nyneq].eqnr=nyneq;
    limt[mini]=1;
    limt[minj]=1;
    trypack=1;
    while(trypack){
    minnew=th+1;
    for(i=0; i<S->neq; ++i){
      if(!limt[i]){
        nnew=estimateNRHSwhenGlued(S->E+i,nyE+nyneq);
        if(nnew==0){
```

```
            tryglue=2;
            printf("Inconsistency Found\n");
            i=S->neq;
          }
          else{
            if(nnew<minnew){
            minnew=nnew;
            mini=i;
            }
          }
        }
      }
      if(tryglue==1 && minnew<=th){//mulig aa putte paa en til
        //printf(" og %d\n",mini);
        glue(nyE+nyneq,S->E+mini,tmp);
        limt[mini]=1;
        deleteEquation(nyE+nyneq);
        nyE[nyneq]=*tmp;
      }
      else
        trypack=0;
      }
      nyneq++;
    }
    else
      tryglue=0;
  }//har pakket saa godt som mulig, kopierer ulimte symboler
  if(tryglue<2){
    for(i=0; i<S->neq; ++i){
      if(limt[i])
      deleteEquation(S->E+i);
      else{
      nyE[nyneq]=S->E[i];
      nyE[nyneq].eqnr=nyneq;
      nyneq++;
      }
    }
    free(S->E);
    S->E=nyE;
    S->neq=nyneq;
    for(i=0; i<S->neq; ++i)
      S->E[i].eqnr=i;
  }
  free(limt);
  for(i=0; i<S->neq; ++i)
    free(gluetab[i]);
  free(gluetab);
  if(tryglue==2)
    return 0;
  return 1;
}
```

4. LINKING.h

```
/* Methods for linking equations in system */

void zeroPrepend(u32 *b, int lb, int nzeros, u32 *x){
  int xw, n0w, lbw, skev, i;

  n0w=(nzeros>>5);
  xw=(lb+nzeros+31)>>5;
  lbw=(lb+31)>>5;
  for(i=0; i<xw; ++i)
    x[i]=0;
  skev=(nzeros&0x1f);
  if(skev){
    for(i=0; i<lbw; ++i){
      x[i+n0w]|=(b[i]<<skev);
      if((i+n0w)<(xw-1))
      x[i+n0w+1]|=(b[i]>>(32-skev));
    }
  }
  else{
    for(i=0; i<lbw; ++i)
      x[i+n0w]=b[i];
  }
}

void zeroAppend(u32 *b, int lb, int nzeros, u32 *x){
  int i, xw, lbw;

  lbw=(lb+31)>>5;
  xw=(lb+nzeros+31)>>5;
  for(i=0; i<lbw; ++i)
    x[i]=b[i];
  for(i=lbw; i<xw; ++i)
    x[i]=0;
}

int collapseCellIDlist(struct bitVector *vl, int nil){
  int bp=0, pp=1;

  while(pp<nil){
    while(pp<nil && v0EqualV1(vl[bp],vl[pp]))
      free(vl[pp++].v);
    if(pp<nil)
      vl[++bp]=vl[pp++];
  }

  return bp+1;
}

int findCellIndex(struct bitVector *vl, int nil, struct bitVector
target){
  int bunn, topp, midt;

  if(v0GreaterThanV1(vl[0],target))//target smaller than smallest in vl
```

```
    return -1;
  if(v0GreaterThanV1(target,vl[nil-1]))//target greater than largest in
vl
    return -1;
  if(v0EqualV1(vl[0],target))
    return 0;
  if(v0EqualV1(vl[nil-1],target))
    return nil-1;

  //Know target is not inside boundary of vl
  bunn=0;
  topp=nil-1;
  while((topp-1)>bunn){
    midt=(bunn+topp)/2;
    if(v0EqualV1(vl[midt],target))

      return midt;
    if(v0GreaterThanV1(vl[midt],target))
      topp=midt;
    else
      bunn=midt;
  }

  return -1;
}

void establishSmallLink(struct eqSymbol *eq0, struct eqSymbol *eq1,
struct linkSymbol *ls, u32 **M0, int commondim){
  /* Spaces spanned by A-matrices in equations overlap in few
dimensions, constructs a link with 2^r cells */
  int i, j, sumnl, sumnlw;
  u32 *x, *ci;

  sumnl=eq0->nlin+eq1->nlin;
  sumnlw=(sumnl+31)>>5;
  ls->nlist[0]=eq0;
  ls->nlist[1]=eq1;
  eq0->link[eq0->nn]=ls;
  eq1->link[eq1->nn]=ls;
  ls->ncells=(1<<commondim);//commondim should be small enough for this
  ls->cellExists=(u8 *)malloc(ls->ncells*sizeof(u8));
  for(i=0; i<2; ++i)
    ls->nCover[i]=(int *)calloc(ls->ncells,sizeof(int));
  eq0->coverID[eq0->nn]=(u32 *)malloc(eq0->originalNRHS*sizeof(u32));
  eq1->coverID[eq1->nn]=(u32 *)malloc(eq1->originalNRHS*sizeof(u32));
  x=(u32 *)malloc(sumnlw*sizeof(u32));
  ci=(u32 *)malloc(sizeof(u32));
  /* Establish connections for eq0 */
  for(i=0; i<eq0->originalNRHS; ++i){
    if(eq0->RHSexists[i]){
      zeroAppend(eq0->b[i],eq0->nlin,eq1->nlin,x);
      Uxb(M0,commondim,sumnl,x,ci);
      ls->nCover[0][ci[0]]++;
      eq0->coverID[eq0->nn][i]=ci[0];
    }
```

```
  }
  /* Establish connections for eq1 */
  for(i=0; i<eq1->originalNRHS; ++i){
    if(eq1->RHSexists[i]){
      zeroPrepend(eq1->b[i],eq1->nlin,eq0->nlin,x);
      Uxb(M0,commondim,sumnl,x,ci);
      ls->nCover[1][ci[0]]++;
      eq1->coverID[eq1->nn][i]=ci[0];
    }
  }
  for(i=0; i<ls->ncells; ++i){
    if(ls->nCover[0][i]>0 || ls->nCover[1][i]>0)
      ls->cellExists[i]=1;
    else
      ls->cellExists[i]=0;
  }
  eq0->nn++;
  eq1->nn++;

  free(x);
  free(ci);
}

void establishBigLink(struct eqSymbol *eqmin, struct eqSymbol *eqmax,
struct linkSymbol *ls, u32 **M0, int commondim){
  /* Constructs a link where the equations overlap in too many
dimensions.  The (number of) cells in the link is based on
     the right-hand sides from the smallest equation */
  int i, j, t, sumnl, sumnlw, lcw, newNcell, cellindex;
  u32 *x;
  struct bitVector ci;

  sumnl=eqmin->nlin+eqmax->nlin;
  sumnlw=(sumnl+31)>>5;
  lcw=(commondim+31)>>5;
  ls->nlist[0]=eqmin;
  ls->nlist[1]=eqmax;
  eqmin->link[eqmin->nn]=ls;
  eqmax->link[eqmax->nn]=ls;
  ls->ncells=eqmin->nrhs;
  if(ls->ncells==0){printf("(establishBigLink)Equations %d and %d
inconsistent\n",eqmin->eqnr,eqmax->eqnr);exit(0);}
  ls->cellExists=(u8 *)malloc(ls->ncells*sizeof(u8));
  ls->cellID=(struct bitVector *)malloc(ls->ncells*sizeof(struct
bitVector));
  for(i=0; i<2; ++i)
    ls->nCover[i]=(int *)calloc(ls->ncells,sizeof(int));
  eqmin->coverID[eqmin->nn]=(u32 *)malloc(eqmin-
>originalNRHS*sizeof(u32));
  eqmax->coverID[eqmax->nn]=(u32 *)malloc(eqmax-
>originalNRHS*sizeof(u32));

  x=(u32 *)malloc(sumnlw*sizeof(u32));
  ci.v=(u32 *)malloc(lcw*sizeof(u32));
  ci.length=commondim;
```

```
  ci.wl=lcw;
  /* Establish connections for eqmin */
  t=0;
  for(i=0; i<eqmin->originalNRHS; ++i){
    if(eqmin->RHSexists[i]){
      zeroAppend(eqmin->b[i],eqmin->nlin,eqmax->nlin,x);
      Uxb(M0,commondim,sumnl,x,ci.v);
      ls->cellID[t].wl=lcw;
      ls->cellID[t].length=commondim;
      ls->cellID[t].v=(u32 *)malloc(lcw*sizeof(u32));
      for(j=0; j<lcw; ++j)
      ls->cellID[t].v[j]=ci.v[j];
      t++;
    }
  }//all different cellIDs, with repetition, created
  if(t!=eqmin->nrhs){printf("(establishBigLink)More existing RHS's (%d)
in eqmin than eqmin->nrhs=%d says!\n",
                        t,eqmin->nrhs);exit(0);}
  mergeSortBitVectors(ls->cellID,ls->ncells);
  newNcell=collapseCellIDlist(ls->cellID,ls->ncells);
  if(newNcell<ls->ncells){
    ls->ncells=newNcell;//the actual number of cells in this link
    ls->cellID=(struct bitVector *)realloc(ls->cellID,ls-
>ncells*sizeof(struct bitVector));
  }
  for(i=0; i<2; ++i)
    ls->nCover[i]=(int *)calloc(ls->ncells,sizeof(int));
  for(i=0; i<eqmin->originalNRHS; ++i){
    if(eqmin->RHSexists[i]){
      zeroAppend(eqmin->b[i],eqmin->nlin,eqmax->nlin,x);
      Uxb(M0,commondim,sumnl,x,ci.v);
      cellindex=findCellIndex(ls->cellID,ls->ncells,ci);
      if(cellindex==-1){printf("(establishBigLink)fant ikke
cellID!");exit(0);}
      eqmin->coverID[eqmin->nn][i]=cellindex;
      ls->nCover[0][cellindex]++;
    }
  }//link to eqmin created
  /* Establish connections for eqmax */
  for(i=0; i<eqmax->originalNRHS; ++i){
    if(eqmax->RHSexists[i]){
      zeroPrepend(eqmax->b[i],eqmax->nlin,eqmin->nlin,x);
      Uxb(M0,commondim,sumnl,x,ci.v);
      cellindex=findCellIndex(ls->cellID,ls->ncells,ci);
      if(cellindex==-1)//RHS i does not agree with eqmin
      deleteRHS(eqmax,i);
      else{
      eqmax->coverID[eqmax->nn][i]=cellindex;
      ls->nCover[1][cellindex]++;
      }
    }
  }//links created, no more use for cellID, deletes it
  for(i=0; i<ls->ncells; ++i)
    free(ls->cellID[i].v);
  free(ls->cellID);
```

```
  ls->cellExists=(u8 *)malloc(ls->ncells*sizeof(u8));
  for(i=0; i<ls->ncells; ++i){
    if(ls->nCover[0][i]>0 || ls->nCover[1][i]>0)
      ls->cellExists[i]=1;
    else
      ls->cellExists[i]=0;
  }
  eqmin->nn++;
  eqmax->nn++;

  free(x);
  free(ci.v);
}

void tryEstablishLink(struct system *S, int eqnr0, int eqnr1){
  struct eqSymbol *eqmin, *eqmax;
  struct linkSymbol *ls;
  int i, commondim, sumnl;// !! deleted var: , sumnlw;
  u32 **U;

  if(S->E[eqnr0].nrhs<S->E[eqnr1].nrhs){
    eqmin=S->E+eqnr0;
    eqmax=S->E+eqnr1;
  }
  else{
    eqmin=S->E+eqnr1;
    eqmax=S->E+eqnr0;
  }//eqmin goes on top when computing U!
  sumnl=eqmin->nlin+eqmax->nlin;
  // !! deleted var: sumnlw=(sumnl+31)>>5;
  U=(u32 **)malloc(sumnl*sizeof(u32 *));
  commondim=computeU(eqmin,eqmax,U);
  if(commondim>0){/* eqmin and eqmax can exchange information,
establish link */
    ls=S->L+S->nlink;
    ls->linknr=S->nlink;
    if(commondim<32 && (1<<commondim)<eqmin->nrhs)
      establishSmallLink(eqmin,eqmax,ls,U+(sumnl-commondim),commondim);
    else
      establishBigLink(eqmin,eqmax,ls,U+(sumnl-commondim),commondim);
    S->nlink++;
  }
  for(i=0; i<sumnl; ++i)
    free(U[i]);
  free(U);
}

void linkSystem(struct system *S){
  /* Establishes connections for all pairs of equations that can
exchange information */
  int i, j, eq0ni, eq1ni;
  struct eqSymbol *eq0, *eq1;
  struct linkSymbol *ls;

  S->nlink=0;
```

```
  for(i=0; i<S->neq; ++i){
    S->E[i].nn=0;
    S->E[i].coverID=(u32 **)malloc((S->neq-1)*sizeof(u32 *));
    S->E[i].link=(struct linkSymbol **)malloc((S->neq-1)*sizeof(struct
linkSymbol *));
  }
  S->L=(struct linkSymbol *)malloc(((S->neq*(S->neq-
1))/2)*sizeof(struct linkSymbol));
  for(i=0; i<S->neq-1; ++i){
    for(j=i+1; j<S->neq; ++j)
      tryEstablishLink(S,i,j);
  }
  S->L=(struct linkSymbol *)realloc(S->L,S->nlink*sizeof(struct
linkSymbol));
  for(i=0; i<S->neq; ++i){
    S->E[i].coverID=(u32 **)realloc(S->E[i].coverID,S-
>E[i].nn*sizeof(u32 *));
    S->E[i].link=(struct linkSymbol **)realloc(S->E[i].link,S-
>E[i].nn*sizeof(struct linkSymbol *));
  }
}

void deLinkSystem(struct system *S){
  /* Removes all links in S and frees the memory */
  int i;
  struct eqSymbol *eq;
  struct linkSymbol *li;

  for(eq=S->E; eq<S->E+S->neq; ++eq){
    if(eq->nn>0){
      for(i=0; i<eq->nn; ++i)
      free(eq->coverID[i]);
      free(eq->coverID);
      free(eq->link);
      eq->nn=0;
    }
  }
  for(li=S->L; li<S->L+S->nlink; ++li){
    free(li->nCover[0]);
    free(li->nCover[1]);
    free(li->cellExists);
  }
  free(S->L);
  S->nlink=0;
}

void makeLinkTab(struct system *S, int **LT){
  int i;

  for(i=0; i<S->neq; ++i)
    LT[i]=(int *)calloc(S->neq,sizeof(int));
  for(i=0; i<S->nlink; ++i){
    LT[S->L[i].nlist[0]-(S->E)][S->L[i].nlist[1]-(S-
>E)]=(int)ceil(log2(S->L[i].ncells));
```

```
    LT[S->L[i].nlist[1]-(S->E)][S->L[i].nlist[0]-(S-
>E)]=(int)ceil(log2(S->L[i].ncells));
  }
}

void printLinkTab(int **LT, int n){
  int i, j;

  for(i=0; i<n; ++i){
    for(j=0; j<n; ++j){
      printf("%2d",LT[i][j]);
      if((j+1)%5==0)
      printf("|");
    }
    if((i+1)%5==0){
      printf("\n");
      for(j=0; j<n; ++j)
      printf("--");
      for(j=0; j<n/5; ++j)
      printf("-");
    }
    printf("\n");
  }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B

(These codes are intended only to be used for experiments with the MRHS method, and neither their author nor the author of this thesis takes any responsibility for their use).

A. GENERATION OF AN EQUATION

```
/*
      aes_eqs.c
      D. Canright 2008 Sep 10 Wed 12:44:56
Generate MRHS Equations for
Small-scale Variants of the AES algorithm
Also does the encryption!
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)
      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 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 <string.h>

#define MAXROUNDS 10
#define MAXROWS 4
#define MAXCOLS 4
#define MAXBITS 8
#define MAXBLOCK MAXROWS*MAXCOLS
#define MAXKEY MAXBLOCK
#define MAXVARS MAXROUNDS*MAXROWS*(MAXCOLS+1)
```

```
#define SR(c,r) ( (c+r) & (nCols-1) )

unsigned char RoundKeys[(MAXROUNDS + 1) * MAXBLOCK];
unsigned char States[(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,
    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 };

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

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

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

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,0xD7,0
xAB,0x76,
0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0
x72,0xC0,
0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0
x31,0x15,
0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0
xB2,0x75,
```

```
0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0
x2F,0x84,
0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0
x58,0xCF,
0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0
x9F,0xA8,
0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0
xF3,0xD2,
0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0
x19,0x73,
0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0
x0B,0xDB,
0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0
xE4,0x79,
0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0
xAE,0x08,
0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0
x8B,0x8A,
0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0
x1D,0x9E,
0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0
x28,0xDF,
0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0
xBB,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.h"    // include 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 (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;
    KeyBits = block * nBits;
    nKeyCols = (nRounds + 1) * nCols;
    nKeyVars = block + nRounds * nRows;
    nVars = nKeyVars + block * (nRounds - 1);
    nEqs = nVars;

    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( void )
{
    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);
    for (i = 0; i < block; i++)
      CT[i] = States[(nRounds+1)*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 )
{
    switch (var) {
    case Key:
      if (round == 0)
          return ( col * nRows + row );
      else   // then  col == 0
          return ( block + (round-1) * nRows + row );
    case X:
      return ( nKeyVars + (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];

    if (argc > 1) {
      sscanf(argv[1], "%1x%1d%1d%1d",
                &nRounds, &nRows, &nCols, &nBits);
    }
    fprintf(stderr, " nRounds=%d, nRows=%d, nCols=%d, nBits=%d,
star=%d\n",
          nRounds, nRows, nCols, nBits, star);
    if (setup())
      fprintf(stderr,
            "Bad parameter(s); now:\n nRounds=%d, nRows=%d, nCols=%d,
nBits=%d, star=%d\n",
            nRounds, nRows, nCols, nBits, star);
//  by default KeyBits = bits in block
      ReadBlock( (argc > 2) ? argv[2] : "", PT);
      ReadBlock( (argc > 3) ? argv[3] : "", 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();
    KeyScheduleEqs();
//  do only single block
    EncryptEqs();

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

    return (0);
}
```

B. MRHS algorithm

```
/*
        mrhs.c
        D. Canright 2008 Aug 22 Fri 13:58:27
Interactive tool to solve MRHS equations

input file format:
header:
      NVAR  # variables
      neq   # equations (symbols)
for each equation symbol:
      nlin  # rows
      nrhs  # RHS
      A     (by rows, binary)
      b     (by cols, binary)

*/

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

int main(int argc, char *argv[])
{
    int i, n;
    char input[101], *filename;
    char menu[] = {
"Enter commands from the menu below.\n"
"  # means a number is required: use nondigit for default value (or for
'all')\n"
   "l   linkSystem(S);\n"
   "a   agreeSystem(S);\n"
   "d   deLinkSystem(S);\n"
   "g # packSystem(S, #); (glue)\n"
   "x   extractLinearInfo(S);\n"
   "w   writeSystem(S);\n"
   "i # EquationInfo(&S->E[#]);\n"
```

```
   "p # printEquation(&S->E[#]);\n"
   "e   printLinEquation(S->linbank, 0, S->linbank->nlin);\n"
   "t   writeLinkTab(S, 36);\n"
   "q   QUIT (exit)\n"
    };

//    struct eqSymbol *tmpeq = (struct eqSymbol *) malloc(sizeof(struct
eqSymbol));
    struct system *S = (struct system *) malloc(sizeof(struct system
));
      CHECKPTR(S);  // test of macro
/*
read in header, set up sys, setup & read eqns
link system
agree
glue
*/

#if 0
      i = 0;
      CHECKPTR(i);  // test of macro
      return(0);
#endif

if ( argc > 1 ) filename = 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 );
  }
printf("got: NVAR = %d (NWORDS = %d); neq = %d\n", NVAR, NWORDS, S->neq
);
printf(menu);
while ( scanf("%100s",input) == 1 ) {
switch (input[0]) {
case 'a':
case 'A':
printf("** agreeSystem\n");
agreeSystem(S);
break;
case 'e':
case 'E':
printLinEquation(S->linbank, 0, S->linbank->nlin);
break;
case 'd':
case 'D':
printf("** deLinkSystem\n");
deLinkSystem(S);
break;
case 'p':
```

```
case 'P':
if ( input[1] ) i=1;
else { scanf("%100s",input); i = 0; }
if ( isdigit(input[i]) ) {
  sscanf( input+i, "%d", &i );
  if ( i < 0 || i >= S->neq ) {fprintf(stderr," bad eq #: %d\n",i);
break;}
  printEquation(&S->E[i]);
  }
else
  for(i=0; i<S->neq; ++i) printEquation(&S->E[i]);
break;
case 'i':
case 'I':
if ( input[1] ) i=1;
else { scanf("%100s",input); i = 0; }
if ( isdigit(input[i]) ) {
  sscanf( input+i, "%d", &i );
  if ( i < 0 || i >= S->neq ) {fprintf(stderr," bad eq #: %d\n",i);
break;}
  EquationInfo(&S->E[i]);
  }
else
  for(i=0; i<S->neq; ++i) EquationInfo(&S->E[i]);
break;
case 'g':
case 'G':
if ( input[1] ) i=1;
else { scanf("%100s",input); i = 0; }
if ( isdigit(input[i]) )
  sscanf( input+i, "%d", &i );
else
  i = 1024;   // arbitrarily picked a (small) max size of glued eqns
printf("** packSystem (glue); max = %d\n", i);
packSystem(S, i);
break;
case 'l':
case 'L':
printf("** linkSystem\n");
linkSystem(S);
break;
case 'q':
case 'Q':
return 0;
case 't':
case 'T':
writeLinkTab(S, 36);
break;
case 'w':
case 'W':
writeSystem(S);
break;
case 'x':
case 'X':
printf("** extractLinearInfo\n");
```

```
extractLinearInfo(S);
break;
default:
printf(menu);
break;
}
printf("now: neq = %d ; nlink = %d ; linbank = %d\n",
 S->neq, S->nlink, S->linbank->nlin );
}

    return (0);
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] W. Trappe and L.C. Washington, *Introduction to Cryptography with Coding Theory*, 2d ed., Upper Saddle River: Pearson Prentice Hall, 2006.

[2] J. Daemen and V. Rijmen, *The Design of Rijndael: AES—The Advanced Encryption Standard,* New York: Springer, 2002.

[3] "Announcing the Advanced Encryption Standard," Federal Information Processing Standards Publication 197, November 26, 2001, [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (accessed June 2008).

[4] C. Cid, S. Murphy and M. Robshaw, *Algebraic Aspects of the Advanced Encryption Standard,* New York: Springer, 2006.

[5] H. Raddum and I. Semaev, "Solving MRHS linear equations," *Cryptology ePrint Archive*, Report 2007/285, August 28, 2007, http://eprint.iacr.org/2007/285 (accessed September 2008).

[6] H. Raddum and I. Semaev (private communication: provided codes in Appendix A).

[7] "Advanced encryption standard," *Wikipedia,* September 2008. [Online]. Available: http://en.wikipedia.org/wiki/Advanced_Encryption_Standard (accessed September 2008).

[8] E. Trichina, L. Korkishko, Secure and Efficient AES Software Implementation for Smart Cards, [Online]. Available: http://eprint.iacr.org/2004/149.pdf (accessed June 2009).

[9] C. Cid, S. Murphy and M.J.B. Robshaw, "Small-scale variants of the AES," Fast Software Encryption: 12th International Workshop (FSE 2005), Paris, France, February 21–23, 2005, *Lecture Notes in Computer Science*, vol. 3557, pp. 145–162, Berlin: Springer, 2005.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor David Canright
   Department of Applied Mathematics
   Naval Postgraduate School
   Monterey, California