2010

High Performance Computing techniques for attacking reduced version of AES using XL and XSL methods

Elizabeth Kleiman
Iowa State University

High Performance Computing techniques for attacking reduced version of AES using XL and XSL methods

by

Elizabeth Kleiman

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Co-majors: Mathematics;

Computer Science

Program of Study Committee:
Clifford Bergman, Co-major Professor
David Fernndez-Baca, Co-major Professor
Maria Axenovich
Giora Slutzki
Srinivas Aluru

Iowa State University

Ames, Iowa

2010

DEDICATION

This thesis is dedicated to my family. To my husband and children for their love and support. To my parents, who have raised me to be the person I am today. Thank you for believing in me along the way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this dissertation. First and foremost, Prof. Clifford Bergman for his guidance, patience and support throughout this research and the writing of this dissertation. I would also like to thank Prof. David Fernndez-Baca, Prof. Giora Slutzki, Prof. Maria Axenovich and Prof. Srinivas Aluru for their efforts and contributions to this work.

I would like to thank Camille Coti for her help on running experiments. Some of the experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

I gratefully acknowledge the help and support provided by Iowa State University HPC Group (Prof. Glenn Luecke, Dr. James Coyle, Dr. Marina Kraeva and Dr. James Hoekstra). I would like to thank Steve Nystrom for his help with Cybluegene.

ABSTRACT

A known-plaintext attack on the Advanced Encryption Standard can be formulated as a system of quadratic multivariate polynomial equations in which the unknowns represent key bits. Algorithms such as XSL and XL use properties of the cipher to build a sparse system of linear equations over the field GF(2) from those multivariate polynomial equations. A scaled down version of AES called Baby Rijndael has structure similar to AES and can be attacked using the XL and XSL techniques among others. This results in a large sparse system of linear equations over the field GF(2) with an unknown number of extraneous solutions that need to be weeded out. High Performance Computing techniques were used to create SPSOLVERMOD2 a parallel software designed to solve sparse systems of linear equations over the field GF(2).

In this thesis we apply XL and XSL attacks on Baby Rijndael. Using SPSOLVERMOD2 we have shown XL and XSL attacks on Baby Rijndael do not give the desired result when one block of message and corresponding cipher text are provided. The number of linearly dependent equations we get close to 100000 and the number of possible solutions is huge. Finally we present the design of SPSOLVERMOD2 as well as the challenges we met on our way. Also the performance results for random matrices on different clusters and supercomputers are discussed.

CHAPTER 1. A General Overview and Introduction

1.1 Introduction

The security of many recent cryptosystems relies on the computational complexity of solving large systems of quadratic multivariate polynomial equations, the MQ problem. The same problem arises naturally in other subareas of Mathematics and Computer Science, such as optimization, combinatorics, coding theory, and computer algebra (3). For instance, several recent papers discuss the connection between the NP-hard MQ problem and the polynomial time problem of solving a sparse system of linear equations over a finite field.

In particular, a known-plaintext attack on the Advanced Encryption Standard (AES) can be formulated in term of an MQ problem, where the unknowns represent key bits. Algorithms such as XSL and XL were introduced to solve this particular MQ problem (3), (4). These algorithms use properties of the cipher to build a sparse system of linear equations over the field GF(2) from multivariate polynomial equations. The exact complexity of these algorithms is unknown. It has also not been determined whether those algorithms are efficient as an attack on AES.

In Chapter 3 a scaled-down version of AES called Baby Rijndael is described. It has a structure similar to AES, and can be attacked using the XL and XSL techniques among others. The XL and XSL attack produce a large sparse system of linear equations over the field GF(2) with an unknown number of extraneous solutions that need to be weeded out. Special software was created to meet this challenge.

This software is implemented using the MPI C language since High Performance computing recourses are required. Gaussian Elimination, which is believed to be reasonably efficient for matrices over GF(2) (18), is the basic technique used by the software. Although there are

different methods to represent large sparse matrices in the computer memory, most of them are inefficient when Gaussian Elimination is used, since the original matrix is constantly evolving and can expand rapidly.

The main challenge of the Gaussian Elimination step is in managing the rapidly increasing size of the matrix — the fill-in problem. Frequently, it is only the equations "owned" by a few of the processors that have grown excessively while other processors still have plenty of free space. To deal with this phenomena a load balance function was developed: it pairs the most loaded processors with ones which have the lightest load, and then exchanges some equations between the pair. Load balancing is time consuming and used only when the processor is close to running out of memory.

Random sparse matrices over field GF(2) have been used to test the program. The performance results with the running time as a function of the size of the system and the number of processors on the different high-performance computers can be found in Chapter 9.

In this dissertation we present results of an XL and XSL attacks on Baby Rijndael. The attack was unsuccessful when one block of plaintext and ciphertext were used. We also describe design and performance of a parallel software called SPSOLVERMOD2. Our research shows that properties of random matrices result in undesirable behavior of SPSOLVERMOD2 and make useless many of supporting functions.

1.2 Thesis Organization

This thesis is divided into three separate parts:

Part 1 is based on my Masters Thesis (17) and presents the Cryptography context of the work. It describes the AES and Baby Rijndael ciphers as well as the XL and XSL methods. Chapter 5 presents an application of the XL and XSL attacks on Baby Rijndael. It also discusses how our cryptography problem can be reduced to a linear algebra problem of solving a large sparse system of equations over the field $GF(2)$.

Part 2 is devoted to the linear algebra aspects of solving the problem above in an efficient way. Chapter 6 describes various methods for solving systems of linear equations. It focuses on

direct methods, in particular on Gauss Elimination process which is used by the solver which we created. Different reordering techniques that make Gauss Elimination more effective are also investigated.

Finally, Part 3 addresses the High Performance Computing aspects of this work. It describes a solver that was developed in order to find all possible solution of the sparse linear systems over the field $GF(2)$. Performance results of this software on different platforms is presented in Chapter 9. The same results are presented in (16) and parts of that paper are reproduced almost verbatim in in this thesis.

PART I

Cryptography: The XL and XSL attack on Baby Rijndael

CHAPTER 2. Rijndael - AES - Advanced Encryption Standard

This chapter is based on my Master Thesis (17).

2.1 Definitions

2.1.1 Cryptography Definitions

There are some very basic concepts in cryptography that we should define in order to understand AES and Baby Rijndael structure.

Definition 2.1.1. A **cryptosystem** is a five-tuple (P, C, K, E, D), where:

1. P is a finite set of possible inputs/plaintexts,
2. C is a finite set of possible outputs/ciphertexts,
3. K is a finite set of possible keys, and
4. for each $k \in K$ there is an encryption function $e_k \in E$, and a corresponding decryption function $d_k \in D$. Each $e_k : P \to C$ and $d_k : C \to P$ has the property $d_k(e_k(x)) = x$ for every plaintext element $x \in P$.

The function e_k is called a cipher. There are many different kinds of ciphers. All of them take a message as an input and return some output. The message can be represented in many ways; it may be just an array of letters or words, or it might be text represented as numbers, or even binary numbers. The output can also constitute an array of letters or numbers.

We will call the input a plaintext block and the output a ciphertext block. The operation of transforming a plaintext block into a ciphertext block is called encryption. The operation of

transforming a ciphertext block into a plaintext block is called decryption. Most of the ciphers use not only some input, but also a key, since it makes the cipher more secure.

To illustrate the concept, assume that Alice wants to send a secret message to Bob. She wants to be sure that nobody except Bob can read the message. This is easy to do if they have some cipher that no one else knows or if they share some secret key. However, in many cases, Bob will never see or speak to Alice, so they won't be able to agree upon such a cipher or a key.

There are two different kinds of ciphers using keys: public-key ciphers and private-key ciphers. The major difference between these two is that in a private-key cipher, only Alice and Bob know the secret key. In a public-key cipher, the key is not secret but rather is common knowledge. We can define these concepts more precisely as follows.

Definition 2.1.2. A **public-key cryptosystem** is a cryptosystem in which each participant has a public key and a private key. It should be infeasible to determine the private key from knowledge of the public key. To send a message to Alice, Bob will use her public key. Nobody except Alice knows her private key, and one must know the private key to decrypt the message.

The most commonly used public key cryptosystem is the RSA cryptosystem. More detail can be found in (25). However, in this paper we will be dealing with AES, also called Rijndael, which is a private-key cryptosystem. Alice and Bob can send the private key for AES using RSA cryptosystem. RSA is much slower than AES this is why it is better to send long messages using AES.

Definition 2.1.3. A **symmetric-key cryptosystem** (or a private-key cryptosystem) is a cryptosystem in which the participants share a secret key. To encrypt a message, Bob will use this key. To decrypt the message Alice will either use the same key or will derive the decryption key from the secret key. In a symmetric-key cryptosystem, exposure of the private key makes the system insecure.

Definition 2.1.4. A **block cipher** is a function which maps n-bit plaintext blocks to n-bit ciphertext blocks. The function is parameterized by a key. n is called the block size.

Definition 2.1.5. An **iterated block cipher** is one that encrypts a plaintext block by a process that has several rounds. In each round, the same transformation or round function is applied to the intermediate result, called the state, using a round key. The set of round keys is usually derived from the user-provided secret key by a key schedule. The number of rounds in an iterated cipher depends on the desired security level. In most cases, an increased number of rounds will improve the security offered by a block cipher.

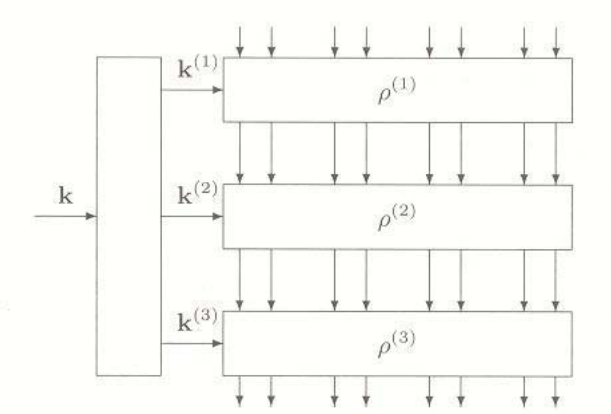


Figure 2.1 Iterative block cipher with three rounds. (5) pg. 25.

Definition 2.1.6. A **key-alternating block cipher** is an iterative block cipher with the following properties:

1. Alternation: The cipher is defined as the alternated application of key-independent round transformations and key additions. The first round key is added before the first round and the last round key is added after the last round.

2. Simple key addition: The round keys are added to the state by means of a simple addition modulo two, called XOR ($\oplus$).

We already mentioned that a block cipher is a function. This is true for any cipher. First, we want to encrypt plaintext, and then we want to decrypt ciphertext to get our plaintext back. So an important condition for our function is that it be one-to-one.

Definition 2.1.7. A function is **one-to-one** if no two different elements in its domain are mapped to the same element in image.

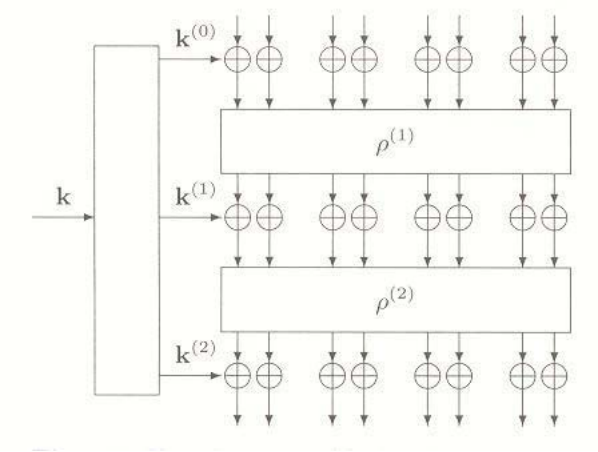


Figure 2.2 Key-alternating block cipher with two rounds. (5) pg. 26.

The purpose of a cipher is to make communication secure. It is not enough to ensure that one cannot crack the cipher you build. The ultimate goal is to ensure that no one else can crack it either. There are many known attacks and one should ensure that the cipher cannot be cracked by any of them, at least not easily.

There is one attack that can be applied to any symmetric-key cipher.

Definition 2.1.8. A **brute-force attack** is a known-plaintext attack. It tries every possible key until one of them "works". That is, knowing the input and output, the attack will try all possible keys until it finds a key K so that the encryption of the input with K gives the desired output.

It is easy to see why this attack will always work. The problem is that the running time of such an attack is typically very large. It can sometimes take centuries! AES can be cracked by Brute-Force in an ideal world, but in real life, based on current technology, it can not.

It should be mentioned that there might be more than one key. That is, if one knows both the plaintext and the ciphertext, there might be more than one key that will encrypt plaintext to this ciphertext. Actually, the false-positive probability for a key is approximately $1 - e^{-2^{m-n}}$, if we know one block of size n, and the key size is m. If we know 2 blocks, then the false-positive probability becomes smaller, namely $1 - e^{-2^{m-2n}}$.

2.1.2 Algebra Definitions

In Section 2.4 we will introduce the way each byte of the input to Rijndael is represented as an element of the field $GF(2^8)$. In order to describe this, we need some basic definitions about fields.

Definition 2.1.9. A **group** is an ordered pair $(G, \star)$, where G is a set and $\star$ is a binary operation on G satisfying the following axioms:

1. $((a \star b) \star c) = (a \star (b \star c))$, for all $a, b, c \in G$.
2. There exists an identity element e such that for all $a \in G$, $a \star e = e \star a = a$.
3. For each $a \in G$, there is an inverse element a^{-1} such that $a \star a^{-1} = a^{-1} \star a = e$.

Definition 2.1.10. A group is an **Abelian group** if for every $a, b \in G$, $a \star b = b \star a$.

Definition 2.1.11. A **field** is a set F together with the binary operations addition $(+)$ and multiplication $(\cdot)$ on F such that:

1. $(F,+)$ is an abelian group with identity 0.
2. $(F - \{0\}, \cdot)$ is an abelian group.
3. For all $a, b, c \in F$, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

$GF(2^8)$ is a field whose elements are polynomials of degree less than 8, with coefficients 0 or 1. In other words, each element of $GF(2^8)$ can be written as:

$$a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

where $a_i \in \{0, 1\}$, for $i = 0, \ldots, 7$.

As we can see from the definition of the field, there are two operations defined on $GF(2^8)$: addition and multiplication. Addition is performed by adding the coefficients for corresponding powers of the polynomials modulo 2. For example,

$$(x^7 + x^5 + x^4 + x^2 + x) + (x^6 + x^5 + x^3 + x^2 + x + 1) = x^7 + x^6 + x^4 + x^3 + 1$$

Multiplication in $GF(2^8)$ is performed by multiplying two polynomials and then reducing this product modulo an irreducible polynomial of degree 8. For example, consider the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. Then

$$(x^6 + x^4 + x^2 + x + 1) \cdot (x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1.$$

Now, take the result mod $m(x)$:

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \text{ (modulo } x^8 + x^4 + x^3 + x + 1) = x^7 + x^6 + 1.$$

Definition 2.1.12. A polynomial is said to be **irreducible** if it cannot be factored into nontrivial polynomials over the same field. For example, over the finite field $GF(2^3)$, the polynomial $x^2 + x + 1$ is irreducible. However, the polynomial $x^2 + 1$ is not, because it can be written as $(x + 1)(x + 1) = x^2 + 2x + 1 = x^2 + 1$.

There is an inverse element for every element of the field except for the 0 polynomial. The existence of unique inverse element comes from the fact that the multiplication is performed modulo an irreducible polynomial. For convenience, we declare that the inverse element of 0 is 0 itself. We will use these facts in the next chapter.

2.2 AES

On January 2, 1997, the National Institute of Standards and Technology (NIST) began looking for a replacement for DES — the Data Encryption Standard. DES was used as an encryption standard for almost 25 years. The decision was made to replace the standard because the key length of DES was relatively small and some new attacks using faster new computers could crack DES. Even a Brute-Force search is possible in this case since for DES only 2^{55} key options exist. It was not secure anymore. The new cryptosystem would be called AES — the Advanced Encryption Standard. It should have a block length of 128 bits and support key lengths of 128, 192 and 256 bits. Fifteen of the submitted systems met these criteria and were accepted by NIST as candidates for AES. All of the candidates were evaluated taking into account three main criteria: security, cost (such as computational efficiency), and

algorithm and implementation characteristics (such as flexibility and algorithm simplicity). There were five finalists, all of which appeared to be secure. On October 2, 2000, Rijndael was selected to be the Advanced Encryption Standard, because its security, performance, efficiency, implementability and flexibility were judged to be superior to the other finalists. Finally, Rijndael was adopted as a standard on November 26, 2001, and posted in the Federal Register on December 4, 2001.

There is only one difference between Rijndael and AES. Rijndael can support block and key lengths between 128 and 256 bits which are multiples of 32. AES only has a specific block length of 128 bits.

2.3 Rijndael Structure

Rijndael is a key iterated and key-alternating block cipher. In this section, we will describe Rijndael with a block size and key size of 128 bits and with 10 rounds.

The cipher has the form

$$E(A) = r_{10} \circ r_9 \circ \cdots \circ r_2 \circ r_1(A \oplus K_0).$$

Every round except the last one has the form

$$r_i(A) = (t \circ \widehat{\sigma} \circ S^*(A)) \oplus K_i,$$

where A is the state and K_i is the ith round key. The last round will not have the t operation. We think about A and K as 4×4 arrays of bytes (one byte is equal to 8 bits).

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \qquad K = \begin{pmatrix} k_{00} & k_{01} & k_{02} & k_{03} \\ k_{10} & k_{11} & k_{12} & k_{13} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{30} & k_{31} & k_{32} & k_{33} \end{pmatrix}$$

Figure 2.3 The matrices A and K.

Every round starts by applying S^* to the state A. More precisely, S^* applies a function s to each byte of A. Function s is a nonlinear, invertible function which takes 8 bits and returns

8 bits. This function is called an S-box and in the Rijndael specification, it is called SubBytes (see Figure 2.6). We will see how this s function works in Section 2.4. For now, we can think about this s function as a table-lookup. We should mention that in some ciphers there is more than one S-box, and different S-boxes are applied on different inputs. In AES the same S-box is applied on all rounds and all inputs.

After computing $S^*(A)$, we apply $\widehat{\sigma}$ to the result, where $\widehat{\sigma}$ is a permutation function called ShiftRow (see Figure 2.7) and is given by

$$\widehat{\sigma}(a_{ij}) = a_{i,j-i(\text{mod}4)}.$$

The next function to be applied is t. This function is called MixColumn (see Figure 2.8). $t(A) = M \cdot A$ where M is a 4×4 matrix from GF(2^8), see Figure 2.4. t is a linear function and $t(A)$ is a 4×4 matrix, the new state. At the end of each round, we will XOR bitwise the state and the round key. This function is called AddRoundKey (see Figure 2.9).

$$M = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

Figure 2.4 The M matrix.

The only part we still need to explain is how to get the round keys. First, we will construct a list $w_0, w_1, \ldots, w_{43}$, each of which is a 4-byte vector, according to the rule:

$$w_i = \begin{cases} (k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}) & \text{for } i\text{=0,1,2,3,} \\ w_{i-4} \oplus w_{i-1} & \text{for } i \bmod 4 \neq 0, \\ w_{i-4} \oplus S^*(\widehat{\alpha}(w_{i-1})) \oplus c_i & \text{for } i \bmod 4 = 0. \end{cases}$$

The function $\widehat{\alpha}$ is defined by $\widehat{\alpha}(x, y, z, u) = (y, z, u, x)$, the c_i's are constant vectors and $k_0, \ldots, k_{15}$ are the bytes of the cipher key. To get K_i for round i, we will just build a matrix with columns $w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}$.

In section 4, we will see a more detailed explanation of an S-box. There are more details about Rijndael structure in (5).

Rijndael has a very interesting property that the structure of the decryption algorithm is similar to that of the encryption algorithm. Both have the same steps, but with inverse functions. In our research we will not use the decryption algorithm at all, so we will not describe it here. A description of the decryption algorithm can be found in (5).

2.4 Rijndael and $GF(2^8)$

In this section, we describe how to represent a byte as an element of the finite field $GF(2^8)$. Let $b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$ represent bits of a byte. Then the corresponding element of $GF(2^8)$ will be $b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$. The addition will just be the addition defined for $GF(2^8)$ and multiplication will be defined modulo the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

In Section 2.3, we said that an S-box is a table-lookup. However, we can actually represent an S-box as a function s acting on elements from $GF(2^8)$. Assume the input to the S-box is some element $a \in GF(2^8)$. We can find the inverse element of a (call it a^{-1}). This inverse element corresponds to multiplication modulo $m(x)$ in $GF(2^8)$. If the S-box function was just defined as an "inverse" function, then it could be attacked using algebraic manipulations. This is why the S-box also uses multiplication and addition. This makes part of the S-box an affine transformation. We denote this affine transformation by f and say that $c = f(d)$, where f is described in Figure 2.5.

$$\begin{pmatrix} c_7 \\ c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} d_7 \\ d_6 \\ d_5 \\ d_4 \\ d_3 \\ d_2 \\ d_1 \\ d_0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Figure 2.5 The affine transformation.

We apply f on a^{-1}, so actually $d = a^{-1}$.

Another way of thinking about an affine function f is as a polynomial multiplication by a fixed polynomial followed by the addition of a constant.

Thus the S-box first finds the inverse element of a and then puts it in the affine transformation. Why did they build the S-box for Rijndael in this way? The affine function was chosen in such a way so that for all S-boxes, there are no fixed points and no opposite fixed points. That is, $s(a) \oplus a \neq 00$ and $s(a) \oplus a \neq \text{FF}$. Where 00 is the zero polynomial of degree 7, and FF is the polynomial of degree 7 with all coefficients equal to 1. In (22), K. Nyberg gives several construction methods for S-boxes. The S-boxes for Rijndael are constructed using one of these methods.

The round function of Rijndael has two main parts: the linear part and the non-linear part. The only non-linear part is SubBytes. We will use the properties to build a system of non-linear equations in Chapter 4 and Chapter 5.

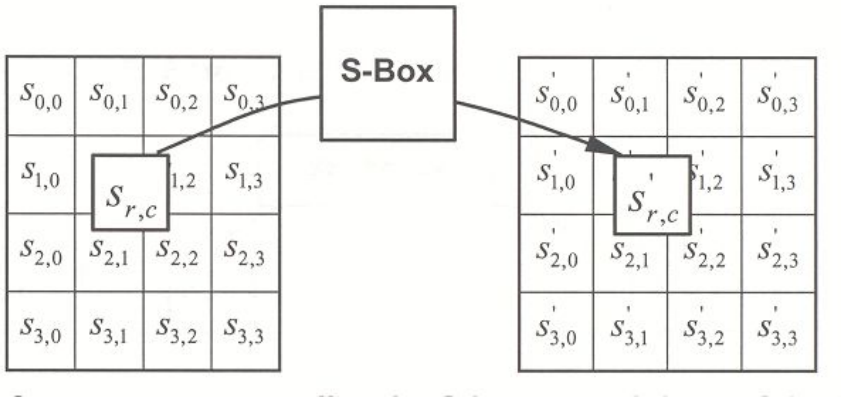


Figure 2.6 SubBytes acts on the individual bytes of the state. (1) pg. 16.

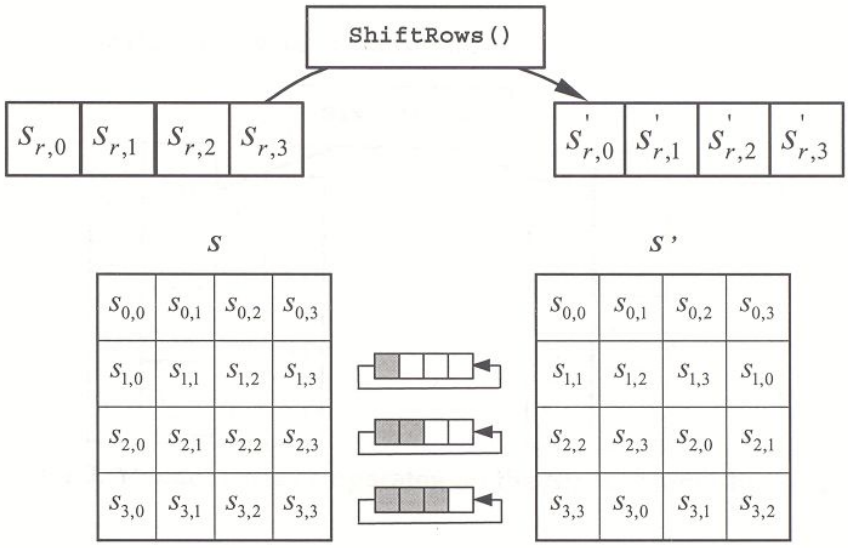


Figure 2.7 ShiftRows operates on the rows of the state. (1) pg. 17.

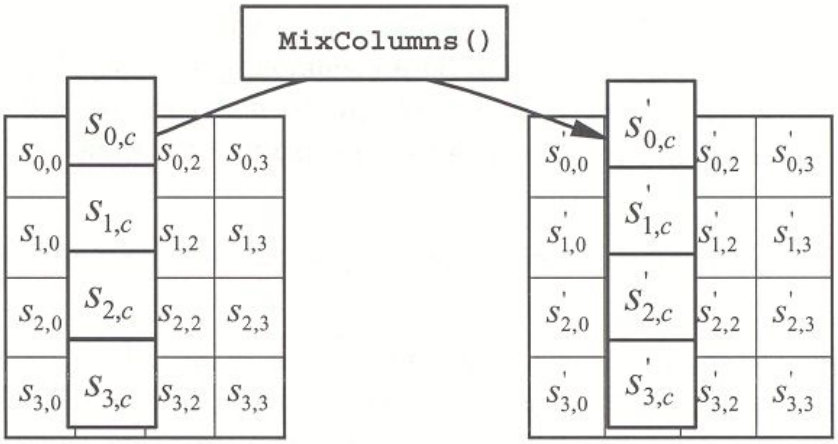


Figure 2.8 MixColumns operates on the columns of the state. (1) pg. 18.

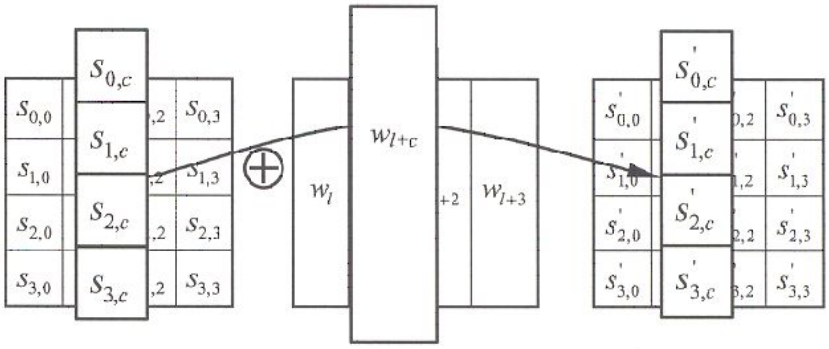


Figure 2.9 AddRoundKey XOR each column of the state with word from the key schedule. (l is a round number). (1) pg. 19.

CHAPTER 3. Baby Rijndael

3.1 Baby Rijndael structure

This chapter is taken almost verbatim from my Master Thesis (17).

3.1.1 Introduction

For the purpose of our research we constructed a new cipher called Baby Rijndael. It is a scaled-down version of the AES cipher. Since Rijndael has an algebraic structure, it is easy to describe this smaller cipher with a similar structure. There were many choices made when Rijndael was constructed, so there is more than one way to build Baby Rijndael.

Baby Rijndael was constructed by Professor Clifford Bergman. He used this cipher as a homework exercise for a cryptography graduate course at Iowa State University. He wanted a cipher that would help his students to learn how to implement Rijndael, but on a smaller, more manageable level.

The block size and key size of baby Rijndael will be 16 bits. We will think of them as 4 hexadicimal digits (called hex digits for short), $h_0h_1h_2h_3$ for blocks and $k_0k_1k_2k_3$ for cipher keys. Note that h_0 consists of the first four bits of the input stream. However, when h_0 is considered as a hex digit, the first bit is considered the high-order bit. The same is true for the cipher key.

For example, the input block `1000 1100 0111 0001` would be represented with $h_0 =$`8`, $h_1 =$ `c`, $h_2 =$ `7`, $h_3 =$ `1`.

Baby Rijndael consists of several rounds, all of which are identical in structure. The default number of rounds is four, but this number can be changed. Changing the number of rounds affects the overall description of the cipher and also the key schedule in a small way. In our

attack, we will use both one-round Baby Rijndael and four-round Baby Rijndael.

The steps of the cipher are applied to the state. The state is usually considered to be a 2×2 array of hex digits. However, for the t operation, the state is considered to be an 8×2 array of bits. In converting between the two, each hex digit is considered to be a *column* of 4 bits with the high-order bit at the top.

The input block is loaded into the state by mapping $h_0h_1h_2h_3$ to $\left(\begin{smallmatrix} h_0 & h_2 \\ h_1 & h_3 \end{smallmatrix}\right)$. For example, the input block `1000 1100 0111 0001` would be loaded as

$$\begin{pmatrix} 8 & 7 \\ c & 1 \end{pmatrix} \text{ which, as an } 8 \times 2 \text{ bit matrix is } \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$$

The state is usually denoted by $\mathbf{a}$.

3.1.2 The cipher

At the beginning of the cipher, the input block is loaded into the state as described above and the round keys are computed. The cipher has the following overall structure:

$$E(\mathbf{a}) = r_4 \circ r_3 \circ r_2 \circ r_1(\mathbf{a} \oplus \mathbf{k}_0).$$

In this expression, $\mathbf{a}$ denotes the state, $\mathbf{k}_0, \mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3, \mathbf{k}_4$ the round keys and

$$r_i(\mathbf{a}) = \big(\mathbf{t} \cdot \hat{\sigma}(S(\mathbf{a}))\big) \oplus \mathbf{k}_i,$$

except that in r_4, multiplication by $\mathbf{t}$ is omitted. At the end of the cipher, the state is unloaded into a 16-bit block in the same order in which it was loaded.

Here is a description of the individual functions of the cipher.

SubBytes: The S operation is a table lookup applied to each hex digit of the state, as shown in Figure 3.1.

$$\begin{pmatrix} h_0 & h_2 \\ h_1 & h_3 \end{pmatrix} \xrightarrow{S} \begin{pmatrix} s(h_0) & s(h_2) \\ s(h_1) & s(h_3) \end{pmatrix}$$

Figure 3.1 SubBytes operation.

where the s function is given by Table 3.1.

Table 3.1 S-box table lookup.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$s(x)$	a	4	3	b	8	e	2	c	5	7	6	f	0	1	9	d

In the next section we will describe how to get this table.

ShiftRows: The $\hat{\sigma}$ operation simply swaps the entries in the second row of the state, Figure 3.2.

$$\begin{pmatrix} h_0 & h_2 \\ h_1 & h_3 \end{pmatrix} \xrightarrow{\hat{\sigma}} \begin{pmatrix} h_0 & h_2 \\ h_3 & h_1 \end{pmatrix}$$

Figure 3.2 ShiftRows operation.

MixColumns: The matrix $\mathbf{t}$ is the 8×8 matrix of bits shown in Figure 3.3.

For this transformation, the state is considered to be an 8×2 matrix of bits. The state is multiplied by $\mathbf{t}$ on the left using matrix multiplication modulo 2: $\mathbf{a} \mapsto \mathbf{t} \cdot \mathbf{a}$.

KeySchedule: At the beginning of the cipher and at the end of each round, the state is bitwise added (mod 2) to the round key. The round keys are 2×2 arrays of hex digits similar to the state. The *columns* of the round keys are defined recursively as follows:

$$w_0 = \begin{pmatrix} k_0 \\ k_1 \end{pmatrix} \qquad w_1 = \begin{pmatrix} k_2 \\ k_3 \end{pmatrix}$$

$$w_{2i} = w_{2i-2} \oplus S(\text{reverse}(w_{2i-1})) \oplus y_i \qquad w_{2i+1} = w_{2i-1} \oplus w_{2i}$$

for $i = 1, 2, 3, 4$. The constants are $y_i = \binom{2^{i-1}}{0}$ and the reverse function interchanges the two entries in the column. Notice that y_i is a vector of length 8 bits and 2^{i-1} and 0 are each four

$$\mathbf{t} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Figure 3.3 MixColumn operation.

bits. The S function is the same as the one used above. Note that all additions are bitwise mod 2. Finally, for $i = 0, 1, 2, 3, 4$, the round key $\mathbf{k}_i$ is the matrix whose columns are w_{2i} and w_{2i+1}.

Baby Rijndael has a structure similar to Rijndael. If we look at a round of Baby Rijndael, it has the same functions as Rijndael. All the functions constructed for Baby Rijndael were built in the same way as the Rijndael functions, but they will work on a smaller state. For example, ShiftRows for Baby Rijndael works in the same way as ShiftRows of Rijndael works on the 2×2 upper left submatrix of Rijndael. The KeySchedule uses the same definition as in Rijndael, but only for 2 smaller w's. In Section 3.2, we will see why the S-box construction of both ciphers is similar.

3.2 Baby Rijndael S-box Structure

We described in Section 2.4 how to represent a byte as an element of finite field $GF(2^8)$. Now we will show how to represent a hex digit as an element of field $GF(2^4)$. Let $b = b_3, b_2, b_1, b_0$. Then the corresponding element of $GF(2^4)$ will be $b_3x^3 + b_2x^2 + b_1x + b_0$ (see Table 3.2). The addition for $GF(2^4)$ will be defined in usual way and multiplication will be defined modulo the irreducible polynomial $m(x) = x^4 + x + 1$.

We will not change the structure of the S-box. It will stay same as for Rijndael, but we can not use the same affine transformation f we had before. Now we have a smaller sate. Let us define a new f for Baby Rijndael in Figure 3.4.

Table 3.2 Different representations for $GF(2^4)$ elements.

hex	binary	polynomial
0	0000	0
1	0001	1
2	0010	x
3	0011	$x+1$
4	0100	x^2
5	0101	x^2+1
6	0110	x^2+x
7	0111	x^2+x+1
8	1000	x^3
9	1001	x^3+1
a	1010	x^3+x
b	1011	x^3+x+1
c	1100	x^3+x^2
d	1101	x^3+x^2+1
e	1110	x^3+x^2+x
f	1111	x^3+x^2+x+1

$$\begin{pmatrix} c_3 \\ c_2 \\ c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} d_3 \\ d_2 \\ d_1 \\ d_0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

Figure 3.4 The affine transformation for Baby Rijndael.

As in Rijndael, we apply f on a^{-1}, so actually $d = a^{-1}$.

Another way of thinking about an affine function f is as polynomial multiplication followed by XOR with a constant. So it can be represented as $s(x) = b(x)g(x)+c(x)$. $g(x)$ is the inverse element of the input to the S-box, $b(x) = x^3 + x^2 + x$ and $c(x) = x^3 + x$. The result will be taken modulo $x^4 + 1$.

So the S-box first finds the inverse element of a and then puts it in the affine transformation. Table 3.3 shows all the inverse elements.

As one can see, our Baby Rijndael S-box has similar properties to the Rijndael S-box. It uses the inverse element and affine transformation and there are no fixed points or opposite

Table 3.3 The inverse elements.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x^{-1}	0	1	9	e	d	b	7	6	f	2	c	5	a	4	3	8

fixed points. This is why we can use Rijndael attacks which uses S-box properties on Baby Rijndael.

3.3 Example

3.3.1 Example for Key Schedule

Sample key expansion for key=6b5d

$$w_0 = \begin{pmatrix} 6 \\ b \end{pmatrix} \qquad w_1 = \begin{pmatrix} 5 \\ d \end{pmatrix}$$

$$w_1 \xrightarrow{\text{reverse}} \begin{pmatrix} d \\ 5 \end{pmatrix} \xrightarrow{S} \begin{pmatrix} 1 \\ e \end{pmatrix} \oplus w_0 = \begin{pmatrix} 7 \\ 5 \end{pmatrix} \oplus y_1 = \begin{pmatrix} 6 \\ 5 \end{pmatrix} = w_2 \qquad w_1 \oplus w_2 = \begin{pmatrix} 3 \\ 8 \end{pmatrix} = w_3$$

$$w_3 \xrightarrow{\text{reverse}} \begin{pmatrix} 8 \\ 3 \end{pmatrix} \xrightarrow{S} \begin{pmatrix} 5 \\ b \end{pmatrix} \oplus w_2 = \begin{pmatrix} 3 \\ e \end{pmatrix} \oplus y_2 = \begin{pmatrix} 1 \\ e \end{pmatrix} = w_4 \qquad w_3 \oplus w_4 = \begin{pmatrix} 2 \\ 6 \end{pmatrix} = w_5$$

$$w_5 \xrightarrow{\text{reverse}} \begin{pmatrix} 6 \\ 2 \end{pmatrix} \xrightarrow{S} \begin{pmatrix} 2 \\ 3 \end{pmatrix} \oplus w_4 = \begin{pmatrix} 3 \\ d \end{pmatrix} \oplus y_3 = \begin{pmatrix} 7 \\ d \end{pmatrix} = w_6 \qquad w_5 \oplus w_6 = \begin{pmatrix} 5 \\ b \end{pmatrix} = w_7$$

$$w_7 \xrightarrow{\text{reverse}} \begin{pmatrix} b \\ 5 \end{pmatrix} \xrightarrow{S} \begin{pmatrix} f \\ e \end{pmatrix} \oplus w_6 = \begin{pmatrix} 8 \\ 3 \end{pmatrix} \oplus y_4 = \begin{pmatrix} 0 \\ 3 \end{pmatrix} = w_8 \qquad w_7 \oplus w_8 = \begin{pmatrix} 5 \\ 8 \end{pmatrix} = w_9$$

Where $y_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $y2 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$ $y_3 = \begin{pmatrix} 4 \\ 0 \end{pmatrix}$ $y_4 = \begin{pmatrix} 8 \\ 0 \end{pmatrix}$

3.3.2 Example for Encryption

Sample encryption for key=`6b5d` and plaintext block=`2ca5`.

round	**start**	**apply S**	**apply $\hat{\sigma}$**	**mult by t**	$\oplus$	**round key=**	
input	$\begin{pmatrix} 2 & a \\ c & 5 \end{pmatrix}$				$\oplus$	$\begin{pmatrix} 6 & 5 \\ b & d \end{pmatrix}$	$=$
1	$\begin{pmatrix} 4 & f \\ 7 & 8 \end{pmatrix}$	$\begin{pmatrix} 8 & d \\ c & 5 \end{pmatrix}$	$\begin{pmatrix} 8 & d \\ 5 & c \end{pmatrix}$	$\begin{pmatrix} 2 & f \\ 0 & 7 \end{pmatrix}$	$\oplus$	$\begin{pmatrix} 6 & 3 \\ 5 & 8 \end{pmatrix}$	$=$
2	$\begin{pmatrix} 4 & c \\ 5 & f \end{pmatrix}$	$\begin{pmatrix} 8 & 0 \\ e & d \end{pmatrix}$	$\begin{pmatrix} 8 & 0 \\ d & e \end{pmatrix}$	$\begin{pmatrix} 0 & a \\ e & 3 \end{pmatrix}$	$\oplus$	$\begin{pmatrix} 1 & 2 \\ e & 6 \end{pmatrix}$	$=$
3	$\begin{pmatrix} 1 & 8 \\ 0 & 5 \end{pmatrix}$	$\begin{pmatrix} 4 & 5 \\ a & e \end{pmatrix}$	$\begin{pmatrix} 4 & 5 \\ e & a \end{pmatrix}$	$\begin{pmatrix} d & 9 \\ 2 & 8 \end{pmatrix}$	$\oplus$	$\begin{pmatrix} 7 & 5 \\ d & b \end{pmatrix}$	$=$
4	$\begin{pmatrix} a & c \\ f & 3 \end{pmatrix}$	$\begin{pmatrix} 6 & 0 \\ d & b \end{pmatrix}$	$\begin{pmatrix} 6 & 0 \\ b & d \end{pmatrix}$		$\oplus$	$\begin{pmatrix} 0 & 5 \\ 3 & 8 \end{pmatrix}$	$=$
output	$\begin{pmatrix} 6 & 5 \\ 8 & 5 \end{pmatrix}$						

Thus the encryption of `2ca5` under key `6b5d` is `6855`.

CHAPTER 4. The XL and XSL attacks

4.1 MQ problem

The problem of a solving linear system of equations with n equations and n unknowns is easy. Gaussian elimination is a well-known algorithm for solving such a system and it has a running time of $O(n^3)$. Actually, depending on the system of equations, there are even quicker techniques.

The MQ problem is the problem of solving a system of multivariate quadratic equations. This means that the equations we will have in our system can have quadratic terms. We will consider a system with m equations and n unknowns with $m > n$, over the field $GF(2)$.

Each equation for the MQ problem can be represented as:

$$\sum_{i,j} a_{i,j,k} x_i x_j + \sum_i b_{i,k} x_i + c_k = 0$$

where x_i's are unknown, $a_{i,j,k}, b_{i,k}, c_k \in GF(2)$ are constant and k indexes the equations we are looking at.

As we already mentioned, a linear system of equations has a polynomial time algorithm, but the MQ problem is NP-hard, see (6). Until now, it was believed that exponential time is needed to solve this problem.

However, for an overdefined systems of multivariate quadratic equations, (that is, $m > n$), some algorithms that appear to run in polynomial time have been proposed. For example, Kipnis and Shamir in (15) presented a new algorithm called relinearization. After their publication, many improved algorithms for relinearization were suggested.

The idea of all these algorithms is simple. We do not know in general how to solve the MQ problem, but we do know how to solve a linear system of equations. So we will transform

multivariate quadratic equations to linear equations.

The problem is that it is not yet clear how these algorithms will work in real life and what running time they will have. It has only been checked on small examples, and there might be cases when those algorithms will not work.

4.2 Relinearization technique

As described above, the idea of linearization is not hard. We simply replace every quadratic term in our system of multivariate quadratic equations by a new variable. That is, assume we have n variables: $x_1, \ldots, x_n$. We will replace every quadratic term $x_i * x_j$ by a new variable y_{ij}. Now we have a system of linear equations, which we know how to solve. However, in order to solve a system of linear equations, we need to have at least as many equations as unknowns. Therefore, there is a big restriction on using this method, assuming that every quadratic term is present, the number of equations we have must be at least $\frac{n^2}{2}$, where n is the number of original variables.

Example 4.2.1: A system of four equations with three unknowns after linearization will become a system of four equations with six unknowns.

$$\begin{aligned} x_1 + x_1x_2 + x_2 &= 0 & \qquad x_1 + y_{12} + x_2 &= 0 \\ x_1x_2 + x_3 &= 1 & \qquad y_{12} + x_3 &= 1 \\ x_2 + x_1x_3 &= 0 & \qquad x_2 + y_{13} &= 0 \\ x_1 + x_2x_3 + x_3 &= 1 & \qquad x_1 + y_{23} + x_3 &= 1 \end{aligned}$$

The relinearization technique was introduced by Aviad Kipnis and Adi Shamir in (15). This method works well for the MQ problem, if the number of equations is at least ϵn^2, where n is the number of variables and $0 \leq \epsilon \leq \frac{1}{2}$. Assume that we have a system of multivariate quadratic equations which meets this requirement. To solve such a system, they suggest first making replacements as in the linearization tecnique. That is, replace every quadratic term $x_i * x_j$, $i \leq j$, by a new variable y_{ij}. Then construct more equations using connections between

the new variables. For example, if $1 \leq a \leq b \leq c \leq d \leq n$, then $(x_a * x_b) * (x_c * x_d) = (x_a * x_c) * (x_b * x_d) = (x_a * x_d) * (x_b * x_c) \Rightarrow y_{ab} * y_{cd} = y_{ac} * y_{bd} = y_{ad} * y_{bc}$.

Now we have more equations, but all the new equations we get have quadratic terms in them. So we will use linearization again to get a system of linear equations. There is a discussion in (15) that argues that this iterative technique will eventually succeed in producing a linear system with sufficiently many equations.

4.3 The XL method for solving MQ problem

XL (which stands for eXtended Linearization) was created by Nicolas Courtois, Alexander Klimov, Jacques Patarin and Adi Shamir in Eurocrypt'2000 (3). The problem they want to solve is described in (3) in the following way:

Let K be a field, and let A be a system of multivariate quadratic equations, $l_i = 0$, $(1 \leq i \leq m)$ where each l_i is the multivariate polynomial $f_i(x_1, \ldots, x_n) - b_i$. The problem is to find at least one solution $x = (x_1, \ldots, x_n) \in K^n$, for a given $b = (b_1, \ldots, b_m) \in K^m$.

In the XL algorithm, we will need to create equations of the form $(\prod_{j=1}^{k} x_{i_j}) * l_i = 0$, where $x_{i_j} \in (x_1, \ldots, x_n)$. Equations of this type are denoted by $x^k l$, and $x^k l$ also denotes the set of all such equations. The set of all terms of degree k is denoted by x^k (so $x^k = \{\prod_{j=1}^{k} x_{i_j} : x_{i_j} \in (x_1, \ldots, x_n)\}$). For $D \in \mathbb{N}$, I_D will be the linear space generated by all the equations of the form $x^k l$ for $0 \leq k \leq D - 2$.

This is how they defined their algorithm.

Definition 4.3.1. The **XL algorithm** executes the following steps:

1. **Multiply:** Generate all the products $(\prod_{i_j}^{k} x_{i_j}) * l_i \in I_D$ with $k \leq D - 2$.

2. **Linearize:** Consider each monomial in x^k of degree at most D as a new variable and perform Gaussian elimination on the equations obtained in 1.

   The ordering on the monomials must be such that all the terms containing one variable (say x_1) are eliminated last.

3. **Solve:** Assume that step 2 yields at least one univariate equation in the powers of x_i for some i. Solve this equation over the finite field K.

4. **Repeat:** Simplify the equations and repeat the process to find the values of the other variables.

In other words, we will have a system of multivariate quadratic equations over the field K represented as a system of equations where each equation looks like

$$\sum_{i,j} a_{i,j,k} x_i x_j + \sum_i b_k x_i + c_k = 0,$$

where the x_i's are unknown, and k is the number of the equation.

To solve this system, we will first agree on some integer $D > 2$. A complexity evaluation of XL in (3) gives the following estimation:

$$D \geq \frac{n}{\sqrt{m}},$$

where m is the number of equations in the original system and n is the number of variables in the original system.

After we pick D, we will take the list of original variables and construct a new list of variables with every possible power less or equal to $D-2$. For example, if the list of variables is (x, y, z) and $D = 4$, then the new list will be $(x, y, z, x^2, y^2, z^2, xy, xz, yz)$. Then we will multiply each original equation by each variable from the new list. This operation will give us more linearly independent equations. It is not necessarily true that all the new equations will be linearly independent, but it is hoped that most of them will be.

Now, let's combine the old and new systems of equations together. We will replace every monomial that appear in any equation by a new variable. For example, the equation $x^4 + x^3y + x^2yz + x^2z^2 + x^2y = 0$ will be $a + b + c + d + f = 0$, where $a = x^4, b = x^3y$, etc. Then we will have a linear system of equations, with the number of equations being larger than the number of variables. This is why we should pick D carefully, so that after multiplication this condition on the number of variables and equations will hold.

Using Gaussian elimination we can find a solution, if one exists. We still have our original variables for the original equations, so we will obtain values for them. It might be that we have more than one solution to the expanded system. For more examples, see (3).

Example 4.2.2: From a system of four equations with three unknowns from Example 4.2.1 we will get a new system of 16 equations with 12 new equations and four original equations. Since we are working over the $GF(2)$ for every x, $x^2 = x$.

$$\begin{aligned} x_1 + x_1x_2 + x_2 &= 0 & x_2 &= 0 \\ x_1x_2 + x_3 &= 1 & x_2 + x_1x_2 + x_2x_3 &= 0 \\ x_1x_3 + x_2 &= 0 & x_1x_2x_3 + x_2 &= 0 \\ x_1 + x_2x_3 + x_3 &= 1 & x_1x_2 + x_2 &= 0 \\ x_1 &= 0 & x_1x_3 + x_1x_2x_3 + x_2x_3 &= 0 \\ x_1 + x_1x_2 + x_1x_3 &= 0 & x_1x_2x_3 &= 0 \\ x_1x_2 + x_1x_3 &= 0 & x_1x_3 + x_2x_3 &= 0 \\ x_1x_2x_3 + x_1x_3 &= 0 & x_1x_3 + x_2x_3 &= 0 \end{aligned}$$

Now we use linearization to get a new system with seven unknowns and 16 equations. Some of the equations may repeat.

$$\begin{aligned} x_1 + y_{12} + x_2 &= 0 & x_2 &= 0 \\ y_{12} + x_3 &= 1 & x_2 + y_{12} + y_{23} &= 0 \\ y_{13} + x_2 &= 0 & z_{123} + x_2 &= 0 \\ x_1 + y_{23} + x_3 &= 1 & y_{12} + x_2 &= 0 \\ x_1 &= 0 & y_{13} + z_{123} + y_{23} &= 0 \\ x_1 + y_{12} + y_{13} &= 0 & z_{123} &= 0 \\ y_{12} + y_{13} &= 0 & y_{13} + y_{23} &= 0 \\ z_{123} + y_{13} &= 0 & y_{13} + y_{23} &= 0 \end{aligned}$$

Using Gauss Elimination we can find a unique solution to the system. The solution will be:

$x_1 = 0,\ x_2 = 0,\ x_3 = 1,\ y_{12} = 0,\ y_{13} = 0,\ y_{23} = 0,\ z_{123} = 0$. It is easy to see that that is the solution of original system too.

A drawback of XL is that in general, we don't know the complexity of the algorithm. However, for proper choices of D it seems to be more efficient than the relinearization method.

In the next chapter, we will see an example of applying this algorithm to one round of Baby Rijndael.

4.4 The XSL attack on MQ problem

The XL algorithm is designed for overdefined systems of quadratic equations and it seems to work for most of them. Some MQ problems have the additional property of being sparse, which means that the equations will be missing many possible quadratic terms. This property can be used to enchance the attack.

XSL (eXtended Sparse Linearization) was created by Nicolas Courtois and Josef Pieprzyk in (4). The cipher was designed to crack XSL-ciphers, ciphers in which the system of equations will be sparse.

Definition 4.4.1. An XSL-cipher is a composition of N_r similar rounds:

X The first round $i = 1$ starts by XORing the input with the session key K_{i-1},

S Then we apply a layer of B bijective S-boxes in parallel, each on s bits,

L Then we apply a linear diffusion layer,

X Then we XOR with another session key K_i. Finally, if $i = N_r$ we finish, otherwise we increment i and go back to step S.

It is easy to see that AES and Baby Rijndael are both XSL-ciphers. For AES, $s = 8$, $B = 4 * N_b$, where N_b is the number of columns in the state. The number of rounds N_r depends on the key size. For Baby Rijndael, we have $N_r = 4$, $s = 4$ and $B = 4$, because in each round we apply 4 S-Boxes.

In the case of the XL attack, the authors of (3) gave the steps in the algorithm. Unfortunately, we don't have such a definition of XSL, because there are many steps in the algorithm

that are not clear yet even to those author. However, we do know that XSL will only work for sparse systems of quadratic equations.

The difference between XL and XSL is that the XL attack will multiply a system of equations by every possible monomial of degree at most $D-2$, where D is fixed. The XSL algorithm suggests multiplying the system of equations only by carefully selected monomials.

XSL will use the system of systems of quadratic equations built for each of the S-boxes. For each S-box, we will have some system of equations.

First fix a constant P. More details about picking P will be given later. In every step we will choose one S-box, and call it active. All the other S-boxes will be called passive for this step. Then we will multiply every equation of the active S-box by all products of $P-1$ monomials arising from the passive S-Boxes. We will repeat until every S-box was active exactly once.

The authors did not present a way of computing an efficient P. If P is very big, then the attack will be similar to the XL method.

If after applying this algorithm we still do not have enough equations to use the relinearization or linearization method, Courtois and Pieprzyk in (4) suggest using the T' method.

After we have applied the XSL method, we will have a new system of equations. Let T be the set of monomials we have in those equations. We want to build still more linearly independent equations. Let T_{x_i} be the set of all the monomials from T that will still be in T when we multiply by x_i. For example, let $T = \{x_1, x_2, x_3, x_1 * x_2, x_2 * x_3\}$. If we work in $GF(2)$, then for every $x \in GF(2)$, $x^2 = x$. Thus $T_{x_1} = \{x_1, x_2, x_1 * x_2\}$.

Build T_{x_1} and T_{x_2}, and apply Gaussian elimination to both. We will think of every monomial as a new variable, and we want to represent every variable in $T - T_{x_1}$ as a combination of variables in T_{x_1} and every variable in $T - T_{x_2}$ as a combination of variables in T_{x_2}. We expect that some subset of this resulting system will contain only terms in T_{x_1}, call it C_1, and some subset will contain only terms in T_{x_2}, call it C_2. We would multiply every equation in C_1 by x_1. After multiplication we will still have only terms from T in this system, so we can substitute, and represent every term in this system as a combination of variables in T_{x_2}. Combine these

new equations and C_2 and multiply each of the equations by x_2. Now we should have some new linearly independent equations. By iterating the process we should get more equations. There is a toy example for the T' method taken from (4) given in Appendix A.

It is not clear how to choose x_1 and x_2, because sometimes this algorithm fails and does not give linearly independent equations. However, the authors of the method suggest that if one system fails, then we should pick some new variables and try the method again. They say that most of the variables should work. More details can be found in (4). Complexity estimates for this attack are also given in the paper.

There are two different kinds of MQ attacks on block ciphers. One of them ignores key schedules, the second one uses key schedules. We should notice that the key schedule of Rijndael uses the same S-boxes as a round of Rijndael. This is why we can build more equations using the key schedule. The XSL attack can be applied in both MQ attacks if the block cipher we use is an XSL-cipher.

CHAPTER 5. The XL and XSL attacks on Baby Rijndael

5.1 The XL attack on one round of Baby Rijndael

5.1.1 Constructing equations

Every cipher can be represented as a system of equations were the unknowns are key bits as well as input and output bit. In this section, we will show how to build these equations for Baby Rijndael. We will build these equations using two different techniques. One way will use the null space equations for the S-boxes. The other way will use the structure of the S-boxes we discussed in Section 3.2.

In this section we will use $X = (x_3, x_2, x_1, x_0)$ to represent the input to an S-box and $Y = (y_3, y_2, y_1, y_0)$ to represent the output of an S-box.

5.1.1.1 Null space equations

To find the null space equations, we will build a 16×37 matrix. Each row will contain the values of 37 monomials: $\{1, x_3, \ldots, x_0, y_3, \ldots, y_0, x_3x_2, x_3x_1, \ldots, x_1x_0, x_3y_3, x_3y_2, \ldots, x_0y_0, y_3y_2, y_3y_1, \ldots, y_1y_0\}$, for each of 16 possible inputs of $\{x_3, x_2, x_1, x_0\}$. See Table 5.1.1.1.

After we build this matrix we can find the null space by row reduction. We use the Mathematica function NullSpace for this. The null space will give us 21 quadratic equations.

$$1 + x_0x_1 + x_0x_2 + x_1x_2 + x_0x_3 + x_2x_3 + y_0y_1 + y_2 + y_3 = 0$$

$$x_1 + x_0x_1 + x_2 + x_0x_2 + x_1x_2 + x_3 + x_0x_3 + x_1x_3 + y_1 + y_0y_2 + y_3 = 0$$

$$1 + x_0 + x_1 + x_0x_1 + x_2 + x_0x_2 + x_0x_3 + y_0y_1 + y_2 + y_3 = 0$$

$$x_0 + x_1 + x_0x_1 + x_2 + x_0x_2 + x_1x_2 + x_3 + y_1 + y_2 + y_3 + y_0y_3 = 0$$

$$x_0x_1 + x_1x_2 + x_3 + x_0x_3 + x_2x_3 + y_0 + y_1 + y_1y_3 = 0$$

$$1 + x_2 + x_0x_2 + x_1x_2 + x_0x_3 + x_1x_3 + y_1 + y_2 + y_2y_3 = 0$$

$$x_0 + x_1 + x_0x_1 + x_2 + x_0x_2 + x_1x_2 + x_3 + x_0x_3 + x_0y_0 + y_1 + y_2 + y_3 + x_3y_3 = 0$$

$$1 + x_1x_2 + x_3 + x_0x_3 + x_1x_3 + y_0 + x_0y_1 + y_2 + y_3 = 0$$

$$x_0x_1 + x_2 + x_0x_2 + x_2x_3 + y_0 + y_1 + y_2 + x_0y_2 + y_3 = 0$$

$$1 + x_0 + x_1 + x_0x_1 + x_3 + x_0x_3 + x_1x_3 + y_3 + x_0y_3 + x_3y_3 = 0$$

$$1 + x_2 + x_1x_2 + x_3 + x_0x_3 + x_2x_3 + y_0 + x_1y_0 + y_1 + y_2 + x_3y_3 = 0$$

$$x_0 + x_1 + x_0x_1 + x_3 + x_0x_3 + x_2x_3 + x_1y_1 + y_2 + x_3y_3 = 0$$

$$x_0 + x_1 + x_2 + x_0x_2 + x_1x_2 + x_3 + x_1x_3 + y_1 + y_2 + x_1y_2 + y_3 = 0$$

$$1 + x_0x_2 + x_1x_2 + x_3 + x_1x_3 + y_0 + y_2 + y_3 + x_1y_3 + x_3y_3 = 0$$

$$1 + x_2 + x_1x_2 + x_0x_3 + x_1x_3 + x_2y_0 + y_1 + y_2 + x_3y_3 = 0$$

$$1 + x_1 + x_0x_1 + x_1x_2 + x_1x_3 + x_2x_3 + x_2y_1 + y_2 + y_3 = 0$$

$$x_1 + x_0x_2 + x_1x_2 + x_3 + x_2x_3 + y_0 + x_2y_2 + x_3y_3 = 0$$

$$1 + x_0 + x_1 + x_0x_1 + x_1x_2 + x_1x_3 + x_2x_3 + y_0 + y_1 + x_2y_3 + x_3y_3 = 0$$

$$1 + x_2 + x_1x_2 + x_3 + x_0x_3 + x_2x_3 + x_3y_0 + y_1 + y_2 = 0$$

$$x_0x_1 + x_2 + x_0x_2 + x_3 + x_1x_3 + y_0 + y_1 + x_3y_1 + y_3 + x_3y_3 = 0$$

$$x_1 + x_0x_1 + x_2 + x_0x_2 + x_1x_2 + x_3 + x_0x_3 + y_1 + x_3y_2 + y_3 + x_3y_3 = 0$$

This yields 21 equations for each S-box. For one round of Baby Rijndael we will have six S-boxes: four S-boxes in the round and another two S-boxes in the Key Schedule. This results in $21 \cdot 6 = 126$ equations. How many unknowns will we have? Each S-box has 4 input bits and 4 output bits, so we will have $8 \cdot 6 = 48$ simple variables.

In addition to these null space equations, we also have plaintext/ciphertext pair we can use. We also know something about the structure of Baby Rijndael, so we can reduce the number of variables that we have. We will do it in Section 5.1.3.

5.1.1.2 Equations with inverse property

As we showed before, an S-box of Baby Rijndael first finds the inverse element of the input and then uses affine transformations. Let $X = (x_3, x_2, x_1, x_0)$ be the input to S-box and let $Y = (y_3, y_2, y_1, y_0)$ be the output of the S-box. The affine transformation is a one-to-one function, so there exists an inverse function, call it h. By applying this inverse function to Y, we will get the inverse element of X. Therefore, we have $h(Y) = X^{-1}$, or $X * h(Y) = (0, 0, 0, 1)$. We can write four equations based on this relationship by equating each of the bits on the lefthand side with a bit on the righthand side. We have a problem if $X = 0$, because then $X * h(Y) = (0, 0, 0, 0)$. This means that our last equation is incorrect, the righthand side should be 0 instead of 1. However, we can still use first three equations.

The equations we get are:

$$
\begin{aligned}
&x_0 + x_2 + x_3 + x_0y_0 + x_1y_0 + x_2y_0 + \\
&\quad + x_3y_0 + x_0y_1 + x_1y_1 + x_0y_2 + x_2y_2 + x_1y_3 + x_2y_3 + x_3y_3 = 0 \\
&x_1 + x_2 + x_3 + x_0y_0 + x_1y_0 + x_2y_0 + \\
&\quad + x_0y_1 + x_1y_2 + x_3y_2 + x_0y_3 + x_1y_3 + x_2y_3 + x_3y_3 = 0 \\
&x_0 + x_1 + x_2 + x_3 + x_0y_0 + x_1y_0 + \\
&\quad + x_3y_1 + x_0y_2 + x_2y_2 + x_3y_2 + x_0y_3 + x_1y_3 + x_2y_3 = 0 \\
&x_1 + x_3 + x_1y_0 + x_2y_0 + x_3y_0 + x_0y_1 + \\
&\quad + x_1y_1 + x_2y_1 + x_0y_2 + x_1y_2 + x_3y_2 + x_0y_3 + x_2y_3 + x_3y_3 = 1
\end{aligned}
$$

In our attack we will use all four equations for each S-box, because for one round it is not likely that $X = (0, 0, 0, 0)$. If we find out that there is no solution for our system of equations, we will erase the "bad" equations from our system and recompute it. We will do the same if the solution we found is wrong; that is, if the solution does not propertly encrypt the given plaintext to the ciphertext. We always can check if the key we found is the correct one by encrypting the plaintext with this key and checking that we get the same ciphertext we have.

For each S-box we will build four equations, so we have another $4 \cdot 6 = 24$ equations (for total of 150 equations) and still $8 \cdot 6 = 48$ variables. We will reduce the number of unknowns

in the next section.

For our convenience, we will rewrite the last equation in the form:

$$x_1 + x_3 + x_1y_0 + x_2y_0 + x_3y_0 + x_0y_1 + x_1y_1 +$$
$$+ x_2y_1 + x_0y_2 + x_1y_2 + x_3y_2 + x_0y_3 + x_2y_3 + x_3y_3 + 1 = 0$$

5.1.1.3 Decrease number of variables

One round of Baby Rijndael can be represented as Figure 5.1. This is why we can represent every input of every S-box as a plaintext XOR an initial key and every output of every S-box as a ciphertext XOR a round key. In this way we can already decrease the number of variables from 48 to $8 \cdot 4 = 32$, the number of bits in the initial key and the round key.

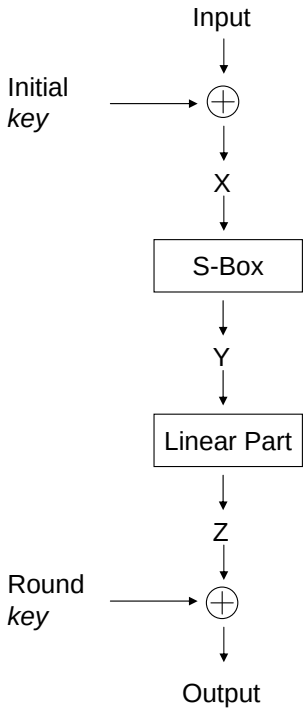


Figure 5.1 One round of Baby Rijndael.

Until now, we have only used the structure of a Baby Rijndael round. We have not used

the KeySchedule yet. This will give us even more information and allow us to further decrease the number of unknowns. Let $K = \begin{pmatrix} k_0 & k_2 \\ k_1 & k_3 \end{pmatrix}$. If we go back to Section 3.1.2, then we see that $w_3 = w_1 \oplus w_2$. This means that the second column of the round key is the XOR of the second column of the initial key and the first column of the round key. In this way we can get rid of 8 more variables, leaving us with only 24 variables.

In the next section, we will solve the system of 150 quadratic equations with 24 variables using the XL attack.

5.1.2 Applying XL attack on equations

In the last section, we ended up with $m = 150$ quadratic equations and $n = 24$ simple variables. We want to solve this system in order to find the secret key. We can't use a linearization method directly because $n^2 = 576$ and $m \not\leq \frac{n^2}{2}$, so we need more equations. We will get these equations by applying the XL attack.

Let's denote the system of equations we have by S. The XL method uses a constant integer $D \geq \frac{n}{\sqrt{m}} = \frac{24}{\sqrt{150}} \approx 1.96$. We want D to be as small as possible, because D will be the degree of the new system, but D must be larger than two, which is the degree of the current system, so we take $D = 3$. This means that we should multiply every equation in S by every original variable, because $D - 2 = 1$.

After this multiplication we get 150*24=3600 equation plus the original 150 equations, so we have a total of 3750 equations with at most $\binom{24}{3} + \binom{24}{2} + 24 = 2324$ monomials. This countsevery possible cubic, quadratic or single term. Now we can use the linearization method. We will replace every monomial by a new variable, but we will not rename the original variables. For example, each $x_i x_j x_k = a_{ijk}, x_i x_j = a_{ij}$ and if we have in some equation just x_i it will remain x_i.

Remember that we are in the field $GF(2)$. This means we have some "rules" that are not true in general, but are true in this field. For example, $x^2 = x, x^3 = x, 2x = 0, 3x = x$, etc. We will apply these rules to make our system easier to solve.

The initial idea was to use the Solve command in Mathematica to solve this linear system

of equations, but it did not work out. It took a lot of time to perform the algorithm and then gave an error message or ran out of RAM. Then we built a 3750×2325 matrix. Each of the first 2324 columns will represent a variable we might have in our system of equations and the last column will represent the constant term. Each row is an equation from our system. To solve the system of equations, we used Mathematica to find the Row Reduced Form of this matrix. It took less than one minute to reduce the matrix.

The matrix in Row Reduced Form had a rank of 2292. This means that we have more than one solution. We actually found four different solutions using the equations. We checked all of them using the plaintext and ciphertext that we had. All four solutions we found encrypted the plaintext we have to the ciphertext we have. So in our case the attack worked pretty well.

5.2 XL and XSL attack on four round Baby Rijndael

5.2.1 Equations for four round Baby Rijndael

We want to represent Baby Rijndael as an MQ problem. Further, we will use the null space equations and properties of the S-boxes. The null space equations will stay the same as they were in Section 5.1.1. One round of Baby Rijndael uses the same S-boxes as four round Baby Rijndael, so again let $X = (x_3, x_2, x_1, x_0)$ be the input to the S-box and let $Y = (y_3, y_2, y_1, y_0)$ be the output of the S-box. Let h be the inverse function of the affine part of the S-box, just as in Section 5.1.2. Then $X * h(Y) = (0, 0, 0, 1)$, unless $X = (0, 0, 0, 0)$.

In four round Baby Rijndael, we have four S-boxes for each round and another eight S-boxes in the Key Schedule, so we have a total of 24 S-boxes. This means that we will have $21 \cdot 24 = 504$ null space equations and another $4 \cdot 24 = 96$ equations from the inverse property. Therefore, we have a total of 600 equations. We have four input bits to each S-box and four output bits for each S-box, which gives $8 \cdot 24 = 192$ variables.

As in Section 5.1.3, we can reduce the number of variables. See Figure 5.2. We can represent each output of an S-box as the input to the next S-box XOR with the round key. We know the plaintext and ciphertext and we can represent the input to the first round S-box as the plaintext XOR the initial key (K_0). The output of the last round S-box will be the ciphertext

XOR the round key (K_4). As in the last chapter, we have only eight (not 16) new bits for each round key, because we can represent the other eight bits using eight bits we have and an initial key or another round key.

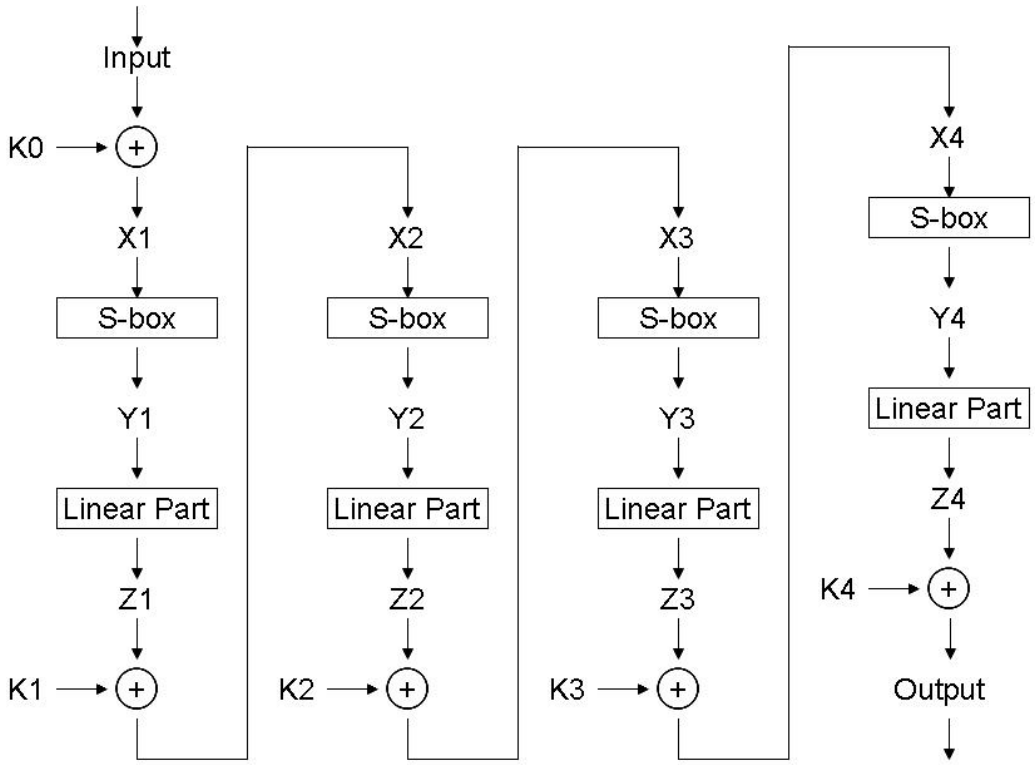


Figure 5.2 Four rounds of Baby Rijndael.

In this way, we can reduce the number of original variables to $n = 96$. If we compute $\frac{n^2}{2}$ we get 4608. We only have 600 equations, so we don't have enough equations to use the linearization method. We need more equations.

First, we will use the inverse property of the S-boxes again. We know $X * h(Y) = (0, 0, 0, 1)$, and this means that $X^2 * h(Y) = X$ and $X * h(Y)^2 = h(Y)$. If we look at this bitwise, we will get eight new equations for each S-box and they are linearly independent of the original inverse equations. Now we have total of $504 + (8 \cdot 24) = 792$ equations. This is better, but still not enough!

After we built the equations, we computed the actual number of monomials we have in the equations, and we had 2332 different monomials. If we compute the possible number of

quadratic and single terms we can have using 96 different variables, we will get $\binom{96}{2} + 96 = 4656$ monomials. We can conclude that we have the sparse property in our equations, many monomials are missing.

5.2.2 The XL method for four round Baby Rijndael

We have a system of $n = 96$ original variables and $m = 792$ quadratic equations. Let's compute the value of D we need to solve such a system using the XL method:

$$D \geq \frac{n}{\sqrt{m}} = \frac{96}{\sqrt{792}} \approx 3.41.$$

Therefore, $D = 4$. This means we should multiply each equation of our system by each possible single term and each possible quadratic monomial. We already computed this number in the last section, the number is 4656. This means we will have a total of $792 \cdot 4656 + 792 = 3,688,344$ equations with $\binom{96}{4} + \binom{96}{3} + \binom{96}{2} + 96 = 3,469,496$ variables. We have more equations than unknowns and we hope the system will have unique solution.

Recall that in Section 5.1.2, we showed that the four equations we get from $X * h(Y) = (0, 0, 0, 1)$ are true only in the case that $X \neq 0$, but we can not be sure that this condition holds. Therefore, it might be better to drop the equation we get using the first bit. Then we will have only three equations for each S-box, and it will reduce our system of equations to $792 - 24 = 768$ quadratic equations. The number of variables will remain the same. The XL method in this case will have $768 \cdot 4656 + 768 = 3,576,576$ equations. We still have more equations than unknowns and we hope to have unique solution.

Actually, when we run the attack the numbers look even more promising. Some of the unknowns we have counted do not appear in the system of equations at all. The number of equations was larger than number of unknowns by almost a million.

Unfortunately, when we ran SPSOLVERMOD2 on this system the result is not as we hoped. Many of equations were linearly dependent and too many equations canceled out. We had more than 2^{100000} possible solutions. Even if we compare it to a brute force attack on original AES, which involves checking 2^{256} possible solutions, the results look really bad. However, the XL

method does not use the sparse property of equations, but the XSL method uses it. In the next section, we use the XSL method to crack Baby Rijndael.

5.2.3 The XSL method for four round Baby Rijndael

5.2.3.1 The XSL method using one block of plaintext

We have a system of $n = 96$ original variables and $m = 792$ quadratic equations, but with only 2332 monomials. For the XL method, we had the parameter D; for the XSL method we need to decide on a parameter P. Unfortunately, we don't have a formula to compute P.

We know that P should be bigger than 1, so let's try to take $P = 2$. Then the algorithm we gave in Section 4.4 says we should multiply every equation of the fixed — active S-box by all products of $P - 1$ monomials arising from all the other — passive S-boxes. We should repeat this multiplication until every S-box is active exactly once.

How many equations will we build and how many monomials will we have? For each S-box we have $21 + 4 + 8 = 33$ equations. Let t_i be the number of monomials in the passive S-boxes when S-box number i is active. Using Mathematica we calculated the value of these t_i's. See Table 5.2.3.1.

We will have 1,807,740 equations. Now we should check how many monomials we have in these equations. We can find an upper bound on the number of monomials by considering what kind of monomials we have in our system of equations. Each monomial has degree of at most four. If we calculate all possible monomials of degree less than or equal to four of 96 variables, we will get 3,469,496. However, we had only 2332 monomials in our original system, and then we multiplied only by these monomials so actually we can represent it as $\binom{2332}{2} + 2332 = 2,720,278$ monomials at most. This number is smaller, but still not good enough. Using Mathematica we found that the actual number of monomials in our system is 1,723,469. This means that we have more equations than variables and the system should be solvable. We need to replace every monomial by a new variable to get a linear system of equations and then we should be able to solve this system. However, when we actually tried to solve the system using SPSOLVERMOD2 the number of possible solutions again was near

2^{100000}.

In the paper (4), the authors mention that for the attack to be successful one should use multiple of blocks of plaintextx and corresponding ciphertext. Recall from Section 2.1 that the false-positive probability for a key is $1 - e^{-2^{m-n}}$ where n is a block size and m is a key size. For Baby Rijndael we used $m = n = 16$. When we use one block of the plaintext we have $1 - e^{-2^{16-16}} = 1 - \frac{1}{e} = 0.63$. It mean that the probability of having at least two keys is very high. This explain to some level why we have so many possible solutions. The system of equations we have, not only contains the keys and state variable, but it also includes the variables we created by multiplication of original variables with monomial. So by expecting at least two possible solutions for key bits we expect many solutions for the whole system.

In the next section we apply XSL attack on four round Baby Rijndael with two blocks of known plaintext and corresponding ciphertext.

5.2.3.2 The XSL method using two blocks of plaintext

Until now we were applying XS and XSL attacks on Baby Rijndael assuming we only know one block of plaintext and corresponding ciphertext. However, a message can have more than one block that is encrypted using the same key. Assume we know two blocks of plaintext and corresponding ciphertext. We know that a message was encrypted using four round Baby Rijndael. Then we can create two sets of equations from Section 5.1.3, one for the first block of the message and another for the second block. Those two sets of equation will share the same key bits.

For one block we had a system of $n = 96$ original variables and $m = 792$ quadratic equations, but with only 2332 monomials. For two blocks we will have $n = 144$ original variable and $m = 1264$ equations (the key variables and key schedule equations are the same for both sets). We will take $P = 2$ again. Then the algorithm discussed in Section 4.4 says we should multiply every equation of the fixed — active S-box by all products of $P-1$ monomials arising from all the other — passive S-boxes. We should repeat this multiplication until every S-box is active exactly once. We apply this attack on each set of equations separately. Then

combine both sets of equations together. We received a system of linear equations with 3909262 unknowns and 10094976 equations.

The false-positive probability for two blocks of Baby Rijndael is $1 - e^{-2^{32-16}} = 1 - e^{-16}$ since block size equal to 32 and the key size is still 16. This probability is quite small for us to hope that a unique key exists. Thus it is reasonable to hope that the system will not have many solutions.

SPSOLVERMOD2 was used to solve this system of linear equations. The system expands rapidly. At some point we have an input with intermediate matrix of 132GB. at this point we were unable to solve the system. However, after performing 3245000 pivots on original matrix, we still have only 3909262 unknowns and 7732113 equations. So we still hope to have a unique solution.

Table 5.1 S-box space matrix.

1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	1	1	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	1	1	0	1	1	0	0	1
1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	0	1	1	0	1	0	0
1	0	1	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	0	0	0	0
1	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	1
1	1	0	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0
1	1	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	0	1	1	0	1	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	1	1	1	0	1	0	0	1	1	1	0	1	0	0	1	0	0	1	1	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	0	0
1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	0	1	0	1	0

Table 5.2 Values of t_i.

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	t_{23}	t_{24}
2300	2300	2312	2300	2256	2268	2248	2212	2272	2284	2240	2228	2288	2224	2224	2252	2304	2304	2332	2332	2304	2332	2332	2332

PART II

Linear Algebra: Methods for solving sparse systems of linear equations

CHAPTER 6. Direct Methods: Gauss Elimination

In Chapter 5 we applied XL and XSL attacks on Baby Rijndael. As a result we obtained large sparse systems of linear equations over the field $GF(2)$ that we needed to solve. Solving linear systems is a well known linear algebra problem and many algorithms exist. This Chapter explores this field and explains in details why we used Gauss Elimination for our purpose.

6.1 Introduction

Finding a solution to a system of linear equations $Ax = b$ is an important and very common problem in linear algebra. The system can have no solutions, a unique solution or many solutions. Many real life problems can be reduced to a problem of solving a system of linear equations (23), (26). Depending on the application it might be enough to find one of the possible solutions of the system, while in others all solutions must be found. In our case we must find all possible solution. Recall from Chapter 5 that in our systems the unknowns represent key bits. Any of the possible solutions can be the one we are looking for. Some of the solutions also can be incorrect for our original system, since linearization adds constrains we don't have in the linear system. For example, we replaced x_1x_2 by a_12. If we get a solution were $x_1 = x_2 = 1$ we must also have $a_12 = 1$. However, we don't have this constrain in linear system, so some of the solutions might be needed to weed out.

The methods for solving linear systems are divided into two classes: direct and iterative. While iterative solvers have become more and more efficient in finding one of the solutions, only direct methods can be used to find them all. Many software packages such as SuperLU (11) and ScaLAPACK (14) use direct methods and are very efficient at solving systems of linear equations. Most of those packages only work for specific classes of matrices that represent the

system, for example symmetric or square matrices. Also almost all of them are designed to operate over the real or complex numbers, but not over a finite field. Solving systems over the reals fundamentally involves approximation. The methods typically rely on the continuity of the underlying field. When operating over a finite field such tools are not available. One must find exact solutions and compute them directly.

As we mentioned before the iterative methods will not work because they find only one possible solution. Some of those methods don't have any randomness in them. The solution they find will be always the same. However, even if we do have randomness and are able to find different solutions by running such algorithm many times, still this will not solve our problem. We need to know how many solutions we have in order to know how many solutions we should find. This can be done only by finding the rank of the matrix, which involves direct methods too.

Our goal is to find all possible solutions of a large sparse systems of linear equations over the field $GF(2)$. We use Gauss Elimination because it is an efficient algorithm for $GF(2)$ (18). We also use reordering process to reduce fill-in.

6.2 Definitions.

First we recall the definition of a field from Section 2.1.2.

Definition 6.2.1. A **field** is a set F together with the binary operations addition $(+)$ and multiplication $(\cdot)$ on F such that:

1. $(F,+)$ is an abelian group with identity 0.

2. $(F - \{0\}, \cdot)$ is an abelian group with identity 1.

3. For all $a, b, c \in F$, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

Both the real numbers and the complex numbers are examples of fields. In this thesis we are concerned with the field $GF(2)$ with underlying set $\{0, 1\}$ and addition given by

$$0 + 0 = 1 + 1 = 0, \quad 0 + 1 = 1 + 0 = 1.$$

Multiplication is the usual product of integers. Note that both addition and multiplication can be thought of as operating on integers modulo 2.

The system of linear equations over the field $GF(2)$ is a system:

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2$$
$$\vdots$$
$$a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m$$

where $a_{ij} \in GF(2)$, for all $i = 1 \dots m, j = 1 \dots n$.

We can form an augmented matrix $[A \mid b]$ for this system:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{bmatrix}$$

Definition 6.2.2. A **row rank** of the matrix is a maximum number of linearly independent rows.

Definition 6.2.3. A matrix is in **row echelon form** if

1. all zero-rows are at the bottom of the matrix and
2. the first nonzero entry in each row (called the *pivot*) is equal to 1 and
3. the pivot occurs to the right of the pivot in the preceding row.

The act of performing a set of row operations to achieve the second condition is called *pivoting.*

Note that the number of non-zero rows in the row reduced form of a matrix is equal to the row-rank of the matrix.

Definition 6.2.4. A matrix is in **row-reduced echelon form** (RREF) if

1. all zero-rows are at the bottom of the matrix and

2. the first nonzero entry in each row is equal to 1 (called leading 1) and

3. the pivot occurs to the right of the leading 1 in the preceding row and

4. in a column containing a leading 1, all other entries of the column are equal to 0.

6.3 Gaussian Elimination

Gaussian Elimination, named after German mathematician Carl Friedreich Gauss, is a common method for solving systems of linear equation. It can be used to solve systems over any field. Given a system of linear equations in matrix form:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

First form an augmented matrix:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{bmatrix}$$

Then apply a sequence of elementary row operations to transform the matrix to the row echelon form (9). The elementary row operations are:

1. Multiply a row by a non-zero member of the field (this operations is useless for matrices over $GF(2)$).

2. Swap two rows.

3. Add c times one row to another for some c in the field (for $GF(2)$ matrices the only choice for c is $c = 1$).

Finally to find the solution use back-substitution to find the elements of the vector x.

The complexity of Gauss Elimination Algorithm is $O(n^3)$. Gauss-Jordan Elimination algorithm is similar to Gauss Elimination algorithm and transforms matrix to RREF. It has the same complexity.

One of the benefits of applying the algorithms on matrices over the field $GF(2)$ is that one's cancel each other. Subtracting pivot row 10100101 from a row 10001101 will result in 00101000. During Gauss Elimination a matrix can increase size. However, the $GF(2)$ property we just discussed hopefully decreases this problem.

Another advantage of using Gauss Elimination for our purpose is that this algorithm can be easily parallelized. Every process is in charge of a set of rows. Given a pivot row every process will perform pivoting on its set of rows. See details about our parallel implementation of the algorithm in Section 8.6.

6.4 Reordering

A sparse matrix is described as a matrix that has very few non-zero elements. One of the main challenges of Gaussian Elimination even when starting with a sparse matrix is the increasing size of the matrix as row operations are applied. This phenomenon is referred to as "fill-in". By performing an initial rearrangement of the rows or columns (or both) of the matrix, one can hope to reduce the degree of fill-in.

Definition 4. Let A be an $n \times n$ matrix and $\pi = \{i_1, i_2, ..., i_n\}$ be a permutation of the set $\{1, 2, ..., n\}$. Then a matrix $A_{\pi,*} = \{a_{\pi(i),j}\}_{i=1,...,n;j=1,...,n}$ is called the *row π-permutation of A* and $A_{*,\pi} = \{a_{i,\pi(i)}\}_{i=1,...,n;j=1,...,n}$ is called the *column π-permutation of A*. Furthermore, $A_{\pi,*} = P_\pi A, A_{*,\pi} = AQ_\pi$ where P_π and Q_π are permutation matrices.

An appropriate reordering applied on the matrix before performing Gaussian Eliminations can decrease fill-in (19), (24). Some of the well known algorithms are Cuthill-McKee (20) and minimum degree reordering (8). Unfortunately, most of these algorithms work only for symmetric matrices. We needed a method which will work for any matrix. A simple reordering can be done by permuting the columns of the matrix in such a way that columns with many non-zero entries are positioned to the right. In this way the earliest pivots will operate on

the sparsest rows. By the time we reach the least sparse columns many of the entries can be canceled out.

PART III

High Performance Computing: SPSOLVEMOD2

CHAPTER 7. Introduction

High Performance computing (HPC) has a broad definition. Most often HPC is associated with a (parallel) supercomputer. HPC is the use of parallel processing for running advanced application programs efficiently, reliably and quickly.

In order to find all possible solutions of a large sparse system of linear equations over the field $GF(2)$ we needed to utilize High Performance techniques. We use parallel computing techniques in order to solve the problem in a quick and efficient way.

There are two common memory models. The first one is the shared memory model where the processors are connected to a global available memory (Figure 7.1). The second one, is the distributed memory model where each processor has its own memory (Figure 7.2).

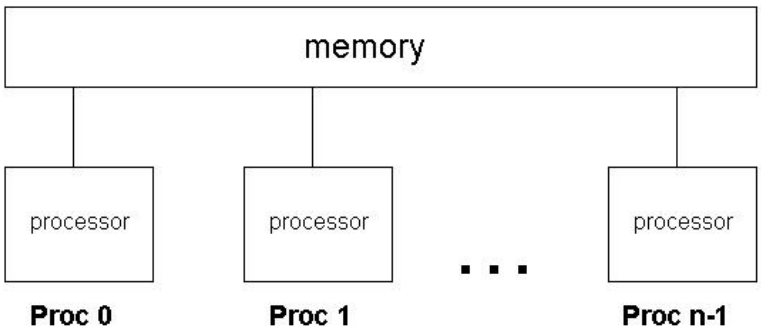


Figure 7.1 Shared Memory Model.

It has become common to have a distributed shared model in which each processor has access to shared memory and also has non-shared private memory.

Most of the systems we had access to were distributed memory systems, so our software was implemented to work in the distributed memory model. The most common parallel language used for this model is the MPI (Message Passing Interface) standard. MPI is a protocol for

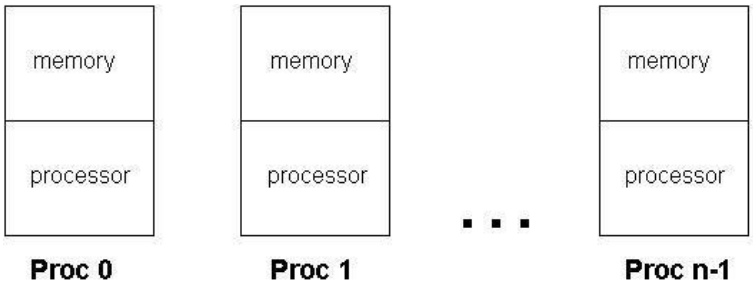


Figure 7.2 Distributed Memory Model.

passing messages between processors that are tied together. MPI has point-to-point messages like MPI_Send and MPI_Recive between two processors as well as collective messages like MPI_Broadcast between all the processors. MPI works with Fortran, C or C++ languages where MPI commands are used for message passing and the serial language is used for serial commands loops, if statements, computations. For our purpose MPI C language was preferred since the structure we used to store the data contained lots of pointers (see Section 8.1 for details).

Parallel programming can be more complicated than serial programming. Concurrency introduces several new problems such as race condition and deadlocks. Race conditions occurs when threads (also called processes, subtasks) try to update the same variable, or write to the same place in output file simultaneously. For example, let thread 0 update x to be 2 and then to print value of x on the screen. Let thread 1 update value of x to be 4. Both threads work concurrently, and thread 0 can print x=2 or x=4 depending on the order of executions.

The deadlock occurs when one thread tries to communicate with another thread that does not expect the communication call. Assume thread 0 wants to send a message to thread 1. Then thread 1 must be waiting to receive the message.

Not all algorithms can benefit from parallelization. Some algorithms are strictly serial because they contain chains of dependent calculations (critical paths). Although, optimally the speed-up from parallelization would be linear; doubling the number of processing elements should halve the runtime. However, most programs have a near-linear speed-up for small numbers of processing elements, which flattens out into a constant value for large numbers of

processing elements.

In the next chapter we provide implementation details of SPSOLVERMOD2. In Chapter 9 we present experimental results for random matrices.

CHAPTER 8. SPSOLVEMOD2 Solver: Implementation Overview

SPSOLVERMOD2 is a parallel software package implemented in the MPI C language. It's primary application is the determination of all possible solutions of a general sparse linear system over the field $GF(2)$. In this chapter we describe the design of the solver.

8.1 General Information

Every system of linear equations can be represented as an augmented matrix where the last column represents the solution vector. From now on we will use the matrix representation. Since our solver works with systems of linear equations over the field $GF(2)$ the matrix can have only 0 or 1 entries.

$$\begin{matrix} x_1 + x_3 = 0 \\ x_1 + x_3 + x_5 = 1 \\ x_2 + x_5 = 0 \\ x_3 + x_4 = 1 \end{matrix} \qquad \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Figure 8.1 Conversion of a system of equations into augmented matrix.

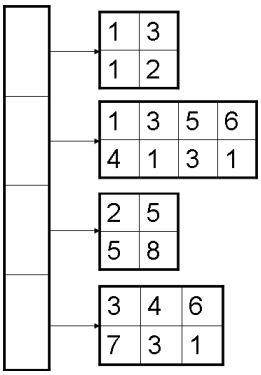
SPSOLVERMOD2 is a parallel solver which takes a matrix that represents a system of linear equations as it's input. It first reorders the matrix, then finds the row reduced echelon form of the matrix and finally finds all possible solutions. The software is divided into thee parts to make it easier to use. When dealing with a smaller system of linear equations, there is no need to apply the reordering algorithm as it does not yield any speed-up, see the results in the last section. Also some of our cryptography matrices ended up having lots of solutions, 2^{100000} or more. In this case, the solution file has size close to $2^{100000} \cdot n \cdot sizeof(int)$ where n is number of unknowns. In that case we don't use the third part of the solver.

8.2 Data Structure: Matrixblock

There are several ways to store a sparse matrix A. It is common to keep only non-zero elements of the matrix, their value and location to reduce the storing space. One of the simplest ways is the so-called coordinate format (24). The data structure consists of three arrays of the same length:

1. a real array containing all the values of the non-zero elements of A in any order;
2. an integer array containing the row index of the corresponding entry in the first array;
3. a second integer array containing the column indices.

Most of the time the elements are listed in row or column order. However, this storage method has a disadvantage: it is hard to add or delete elements from the matrix. During the pivoting process the matrix changes constantly. This is why we used a linked list format with some modifications. An array of size equal to the number of rows, points to a list corresponding to the row. Each element of the list stores the position and value of an element in a row.



Since we work over the field $GF(2)$ there is no need to store the value of the matrix entry. It is always equal to 1. This structure gives us flexibility to change only a small part of the data structure when a row is modified.

In the distributed memory model, every thread can only see his data. It is memory consuming to store the whole matrix on each thread. This is why we divide the matrix into np (number of processors) separate blocks. Each block contains almost the same number of equation $\frac{\#equations}{np}$. Each thread stores one Matrixblock, where the first thread keeps information

about the first $\frac{\#equations}{np}$ matrix rows in it's Matrixblock, the second thread keeps information about the next $\frac{\#equations}{np}$ matrix rows in it's Matrixblock, and so on.

Our structure, Matrixblock, also has a couple of supporting arrays. An array that stores the first element in each row and an array that stores the location and value of the current element. The current element is an element of the row that is closest to the current pivot (must be equal or larger than the pivot). Below is the definition of Matrixblock from the code.

```
struct {
  int nAllEquations;          // Number of equation
  int nAllUnknowns;           // Number of unknowns
  int firstEquation;          // Number of the first equation
                                 stored on processor
  int nEquations;             // Number of equations stored on processor
  int *nUnknownsByEquation; // Array, nUnknownsByEquation[i] is number
                                 of nonzero unknowns in equation i
  int **unknownsByEquation; // Array of pointers, unknownsByEquation[i]
                                 points to array representing equation i
  long nUnknowns;
  int *currentIndex;          // Array, for each equation it gives an
                                 index of an element used the last
  int *currentValue;          // Array, for each equation it gives a
                                 value of an element used the last
  int *firstValue;            // Array, for each equation it gives a
                                 value of the first unknown in it
}
typedef matrixBlock;
```

8.3 I/O

The systems of linear equations derived from our cryptography problem were very large. The size of the input files was close to 1GB. For this reason we wanted efficient I/O procedures. Read and write functions were implemented using MPI I/O routines. The read function reads the input file and stores it as the data structure described above. The write function stores data from the structure into an output file. The output and input files have the same format. The write function is also used when an intermediate matrix should be saved. This turned out to be very handy when we encountered the walltime limits.

Unfortunately, lightning (one of the machines we used) had only one file infiniband. This made parallel I/O functions useless. Actually those routines cause trouble for the file system. This is why a different version of I/O was written using only serial reads and writes.

In order to save space and make I/O more efficient our solver takes as input a binary file. The file contains the dimensions of the matrix followed by the number of non-zero entries in each row. Finally, the file includes row-by-row positions of the non-zero entries. See Figure 8.2 for an example. This helps to calculate offsets for the read commands. Each thread get the same number of rows (the number may differ by one). Every row is stored only on one thread. A thread first calculates which rows are his, then where the information about his rows is stored. This can be easily done by first reading the number of elements in each of his rows. Sum the non-zero elements on every thread and distribute the information between the threads using MPI_Scan routine. The MPI_Scan will return the amount of non-zero elements stored on all threads with smaller ranks.

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \qquad 4\ 6\ 2\ 4\ 2\ 3\ 1\ 3\ 1\ 3\ 5\ 6\ 2\ 5\ 3\ 4\ 6$$

Figure 8.2 Matrix and corresponding input file.

After we finish reading all the equations, we close the file and update supporting arrays. At the beginning, all these arrays point to, or store, the value of the first non-zero element

in each equation. By the end of I/O read function, the Matrixblock structure on each thread is filled with information and ready for the next step. This step can be reordering, finding the row reduced echelon form or constructing the solutions. See Figure 8.3 for example of Matrixblocks constructed using an input file.

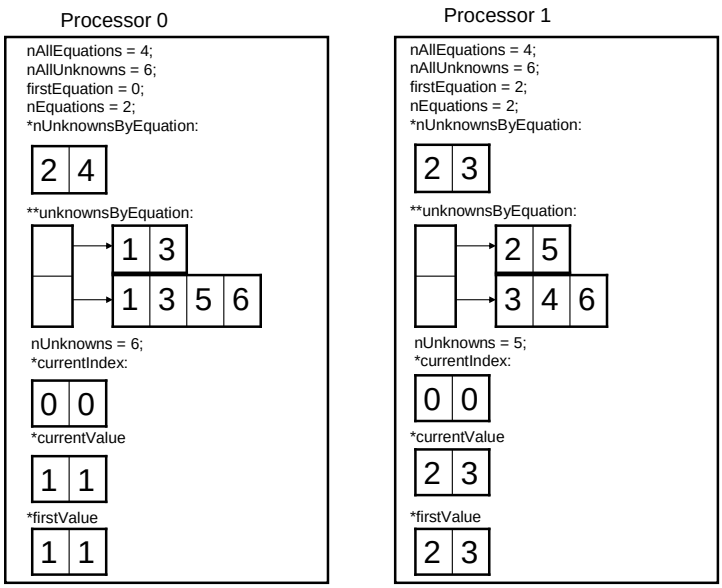


Figure 8.3 Matrixblock for input file from Figure 8.2 and 2 processors.

The write function works in a similar manner. At the completion of the write function a new file with so called state matrix is created. The output and input files have the same format, so if needed the output file can be used as input file for SPSOLVERMOD2.

Implementing I/O so that each thread only reads/writes its own information, saves quite a bit of time. In the Experimental Results section it is shown that for many random matrices the reading and writing parts take only seconds. When our original cryptography matrices where checked, the reading time for an approximately 1GB matrix took less than a minute, while writing an approximately 8GB output file took an hour.

8.4 Reorder

We have discovered that without some preprocessing, Gaussian elimination, even on a very sparse matrix, is prone to considerable fill-in. For example a $5,000,000 \times 5,000,000$ matrix containing approximately 32 entries/row grew 8-fold after only $150,000$ pivots. Row operations on such large matrices becomes very CPU-intensive. For this reason we decided to implement the following simple reordering scheme.

1. Build a table containing, for each column, the column number and the number of non-zero entries in each column (the *column count*).

2. Sort the table on the column count (use quick sort).

3. Reorder/rename the columns so that the sparsest columns appear first.

In this way the earliest pivots operate on the most sparse columns. One hopes that by the time we reach the least sparse columns, many of the entries will have canceled out. This approach has shown good results for our cryptography matrices. While we were unable to find the row reduced form of a $5,000,000 \times 5,000,000$ matrix containing approximately 32 entries per row even in a couple of 12 hour runs on 128 threads, after reordering the same task was achieved in less than 12 hours.

8.5 Load Balance

We begin by evenly distributing the rows to the various threads. What frequently happens is that after a number of pivots, the rows owned by one thread have grown considerably, while those of another thread, not at all. The result is that the first thread runs out of memory while the second's memory is underutilized.

To address this problem, a load-balancing algorithm was devised. We shall refer to the total number of nonzero entries in all of the rows assigned to a particular thread as its "thread-count". An upper threshold (u) and a lower threshold (ℓ) are set. Whenever the thread-count on at least one thread exceeds u load-balancing is performed. Every thread with thread-count

larger than ℓ is paired with a thread whose count lies below ℓ. The two threads exchange rows in an effort to balance their thread-counts, see Figure 8.4. When load-balancing, we pair the most loaded thread with the least loaded, etc. If we run out of threads with counts below ℓ, we report that load-balancing is impossible.

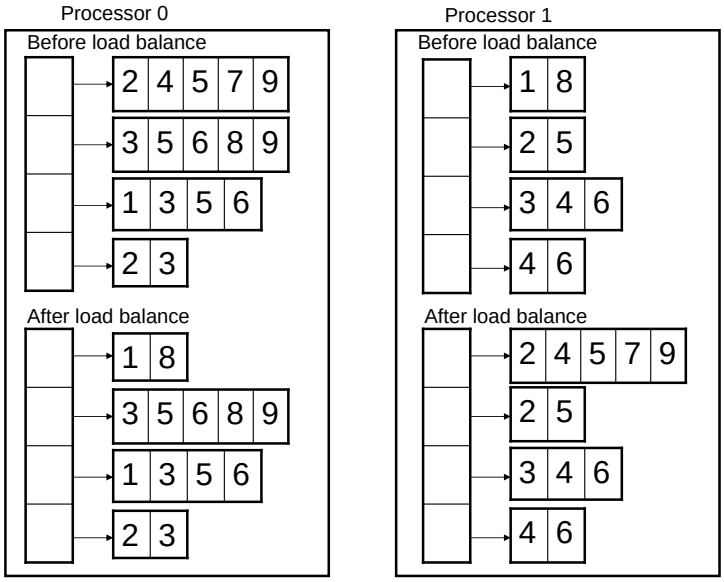


Figure 8.4 Load Balance.

Since load-balancing is time-consuming, the threshold u should be set fairly high so that the algorithm is invoked only when truly needed. One subtle point to note: when load-balancing, rows are *exchanged* between threads. By doing this, the number of rows per thread remains constant. There is a reason for this. If, for example, one thread owned 5 very dense rows while a second had 100 sparse rows, when performing a row operation there is a good chance that the first thread will wind up waiting for the second to finish.

8.6 Gauss Elimination

Gaussian elimination is designed to transform an arbitrary matrix into row-echelon form. As we explained earlier, the rows of the matrix are distributed uniformly among the threads. Let k denote the number of rows per thread. At the start of the process, (before any load-

balancing) thread 0 will own the first k rows, thread 1, the next k, etc.

In the usual approach to Gaussian Elimination one starts with row 1 and applies row operations in order to eliminate all entries in column 1 below the pivot. After this, row 1 column 1 is a pivot element and we can ignore column 1 for the remainder of the algorithm. Then one moves on to row 2, column 2 and repeats the process.

However if one implements this standard approach on our parallel framework, one finds that after k pivots, thread 0 has nothing to do. For this reason we instead reduce our matrix to what we shall call "unordered reduced row-echelon form". That is, instead of working top-to-bottom, we choose the pivot row according to a different strategy (we still choose the pivot column from left-to-right). Also we eliminate all other entries in the pivot column, not just those below the pivot. This strategy reduces communication costs.

Below is a description of the algorithm. Note that since it is a parallel algorithm every thread executes all lines.

```
for(pivot=1;pivot<#unknowns;pivot++){
    1. Among all possible pivot rows owned by you, choose one
       with the smallest number, Rt, of nonzero entries.
    2. Distribute Rt to all other threads.
    3. Choose a thread tmin with minimal value of Rt.
    4. Thread tmin broadcast pivot row to all threads.
    5. for(j=0;j<k;j++){
         If there is 1 in column pivot, eliminate it using pivot row.
       }
}
```

In step 1–3 we chose a pivot row. We do it using a supporting array that stores value of the first element in the row. In step 5 we need to check if a row has a 1 in pivot column. This is easily done using a supporting array that stores each row's current element. We update those arrays when we eliminate the 1 in the pivot column.

Although, it is easy to say "eliminate 1 from row r", in real life such a routine has many *if*

statements. *If* statements always take time. In order to decrease the amount of *if*s every row has an extra element at the end with a large value (think of it as ∞). By using this trick there is no need to check if either the pivot row or row r has elements left.

Here is how it works. Lets call the pivot row pr, where r is a row we want to eliminate 1 from and nr will be a new row we are building. Call function with $i = 0, j = 0$ and $k = 0$.

```
function(i,j,k,pr,nr,r){
  if(pr[i]<r[j]){
    nr[k]=pr[i];
    return function(i+1,j,k+1,pr,nr,r);
  }
  else{
    if(pr[i]>r[j]){
      nr[k]=r[j];
      return function(i,j+1,k+1,pr,nr,r);
    }
    else{
      if(pr[i]==infinity){
        nr[k]=infinity;
        return new row;
      }
      else
        return function(i+1,j+1,k,pr,nr,r);
    }
  }
}
```

8.7 Finding the Solutions

The last part of SPSOLVERMOD2 finds all possible solutions of the system of equations over the field $GF(2)$. Given a matrix in unordered row reduced echelon form the routine:

1. Check if solution exits — look for 0=1 equation. If such an inconsistency exists, print "NO SOLUTION".

2. Check if only one solution exists — use the rank. If so, find the solution and store in solution output file.

3. If more then one solution exists:

   (a) Find all the free variables and create a vector that will hold an assigned value. Think of the vector as a binary number.

   (b) Start with assignment of all zeros to free variables (binary number 0).

   (c) Calculate solution and store it in solution file.

   (d) Change the assignment by adding 1 to binary number. Repeat c until all free variables have value 1.

The solution file has the number of solutions, length of each solution, and then the solutions one after another. The running time of this part depends on number of possible solutions.

CHAPTER 9. SPSOLVEMOD2 Solver: Experimental Results

9.1 Experimental Platforms

Experiments presented here were carried out on four typical platforms: ISU's Blue Gene/L supercomputer CyBlue (10), ISU's Lightning (12) AMD Opteron cluster, ISU's LightningSMP (13) AMD Opteron cluster of SMP nodes and the GdX cluster from the Grid'5000 experimental grid (2).

CyBlue has 1024 nodes, each node containing two dual-core PPC 440 CPUs running at 700Mhz, with 512MB of RAM per node. We compiled the application using GCC compiler with the -O3 optimization flag. CyBlue uses a custom version of MPICH as its MPI communication library. The system uses FC for the high speed interconnect.

Lightning is an AMD Opteron cluster with 592 processor cores (148 nodes) with each node having 8 GBytes of memory (1182 GB total memory on the cluster). Nodes are interconnected with a high performance InfiniPath HTX communication network for MPI communication and a Gigabit Ethernet switch for I/O, management, and PBS communication. The communication library we used is MPICH and the nodes are booted under Linux 2.6.9-42ELsmp. We compiled the application using the Pathscale compiler with optimization options -O2 -OPT:wrap_around_unsafe_opt=OFF.

LightningSMP is an AMD Opteron cluster of SMP nodes. We used 16-core nodes (quad processor, quad core) with 64 GB of memory each. Nodes are interconnected with an Infiniband communication network for low latency/high bandwidth MPI communication and a Gigabit Ethernet switch for I/O, management, and PBS communication. The communication library we used is OpenMPI (7) and the nodes are booted under Linux 2.6.18-128.1.10.el5. We compiled the application using Pathscale with -O2 optimization flag.

The last platform we used is the Grid'5000 experimental testbed, a dedicated reconfigurable and controllable experimental platform featuring 13 clusters, each with 58 to 342 PCs, interconnected through Renater (the French Educational and Research wide area Network). We used up to 256 nodes from the GdX cluster that features two AMD Opteron 246 and 250, running at 2.0 and 2.4 GHz. Nodes are interconnected through a GigaEthernet network and booted under Linux 2.6.26. The MPI library we used is Open MPI 1.3. We compiled the application using GCC 4.2 with -O3 optimization flag.

9.2 Random Matrices

In order to evaluate the performance of the software, a set of test matrices was required. For this Dr. Bergman generated a series of random, sparse matrices over $GF(2)$ of various sizes and ranks. The *rank* of a matrix is the number of pivot elements in its row-echelon form. The *co-rank* (also known as the nullity) of a matrix is the difference between the number of columns and the rank. A system $Ax = b$ over $GF(2)$ will have either no solutions or 2^c solutions, where c is the co-rank of the matrix A.

Our strategy for generating a random $m \times n$ matrix over $GF(2)$ with co-rank c and $m > n$ is quite simple.

1. Create an $m \times n$ matrix with 1's on the first $n - c$ diagonal entries and 0's elsewhere.
2. Randomly insert 1's to the right of the $n - c$ diagonal entries.
3. Repeatedly choose, at random, distinct indices i and j and add row i to row j.
4. Finally apply a random permutation of the rows of the matrix and of the columns of the matrix.

We do not claim that this procedure will generate every matrix (with co-rank c) with equal probability. We must balance the uniformity of the distribution with execution time.

In step 2, the probability, p, that an entry will be set to 1 governs the sparseness of the resulting matrix. The number of times step 3 is applied will also affect the sparseness, but to

a lesser degree. Let us denote by δ the average number of nonzero entries per row. To attain a target δ in the final matrix, one should take p to be slightly less than $2(\delta - 1)/n$. We chose to execute step 3 $2\sqrt{m}$ times. A larger number might result in a more random behavior.

For added efficiency, we performed step 2 as follows. For each $i < n - c$ choose an integer t distributed according to a Poisson distribution with expected value $p(n - i)$. Then choose t-many column numbers uniformly, at random, from $\{i+1, i+2, \ldots, n\}$. These are the entries in row i that get set to 1.

9.3 Performance results for small random matrices

We investigate the performance of the algorithm on random, sparse matrices of varying sizes and densities. This subsection presents results for random matrices of approximate sizes 50000×50000 and 100000×100000 called matrices 1 and 2. The matrices have approximately $1,352,000$ and $2,691,233$ non-zero entries respectively. Six matrices of each size were created. Two matrices had 1 possible solution, two matrices had 32 possible solutions and two had 256 possible solutions. We ran experiments on four experimental platforms: lightning, lightningSMP, Grid5000 and Cyblue. Then we calculated the average running time for matrices 1 and 2. Figure 9.1. presents a chart of the runs using 32, 64 and 128 threads. Figures 9.2 and 9.3 are tables of same results for matrix 1 and 2 with more details. They divided into four parts: I/O time, finding the row reduced echelon form time, finding the solution time and total time. We did not apply the reorder algorithm on those matrices since they have small size.

We can see from Graph 2 that lightning and lightningSMP show the best performance. Our guess is that the time difference can depend on the compiler. Both lightning and lightningSMP use the Pathscale compiler, while Grid5000 and Cyblue use GCC compiler. Actually, we also tried using the IBM compiler on Cyblue, however, the results were much worse.

By taking a look at the tables in Figure 9.2 and 9.3, we can see that increasing the number of threads improves performance. The only exception is Cyblue, where a change from 64 to 128 gave almost no improvement at all. We ran several additional test on the supercomputer and got similar results using 256 and 512 threads. The results are presented at section IV.D.

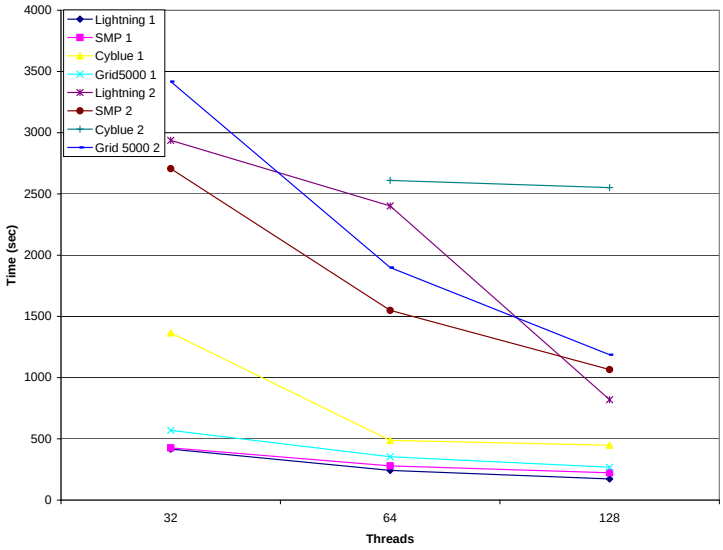


Figure 9.1 Matrices 1 and 2.

We used a serial I/O function, which gives promising results for clusters. The Cyblue I/O results are not as impressive. We tried to use our MPI I/O functions on Cyblue, but there was very little improvement.

9.4 Performance results for large random matrices

This subsection presents results for random matrices of approximate sizes 500000×500000, 1000000×1000000 and 5000000×5000000 called matrices 3, 4 and 5. The matrices have approximately 2255000, 4407727 and 20006120 non-zero entries correspondingly. Three matrices of each size were created: one with 1 possible solution, one with 32 possible solutions and one with 256 possible solutions. As before we ran experiments on four experimental platforms: lightning, lightningSMP, Grid5000 and Cyblue. Cyblue has 512MB of RAM per node and was unable to construct the necessary data structure from the input files. The malloc function used to allocate space for some of the rows returned with an error. This is why the chart displays only the results for three of the machines.

To construct the chart in Figure 9.4, we took average running times from three matrices of each size. We used 32, 64 and 128 threads. Our results show that the reordering algorithm has

		32	64	128
lightning	i/o read&write	0.970581	2.52989	1.71174
	find RREF	413.7534	237.9963	170.0918
	find solution	1.556988	2.052904	1.09743
	Total:	**416.281**	**242.5791**	**172.901**
SMP	i/o read&write	12.75897	9.840824	16.42543
	find RREF	405.7852	263.6354	200.9423
	find solution	8.175268	6.049766	5.507079
	Total:	**426.7194**	**279.526**	**222.8748**
Grid5000	i/o read&write	2.323967	2.871135	4.135979
	find RREF	565.7307	349.2106	262.2239
	find solution	1.434815	1.565878	2.313098
	Total:	**569.4894**	**353.6476**	**268.673**
Cyblue	i/o read&write	69.21224	93.32425	68.18037
	find RREF	1294.392	380.4477	378.9946
	find solution	1.646677	14.17056	1.315804
	Total:	**1365.251**	**487.9425**	**448.4907**

Figure 9.2 Matrix 1.

		32	64	128
lightning	i/o read&write	1.718309	6.01546	3.098082
	find RREF	2933.446	2392.503	813.8769
	find solution	2.746131	3.511424	2.889621
	Total:	**2937.911**	**2402.03**	**819.8646**
SMP	i/o read&write	2.757154	4.025831	51.95488
	find RREF	2700.099	1542.307	1007.293
	find solution	4.094313	3.30478	6.389119
	Total:	**2706.951**	**1549.638**	**1065.637**
Grid5000	i/o read&write	3.56138	2.249488	3.99873
	find RREF	3411.184	1894.51	1179.612
	find solution	2.615994	2.345409	3.213319
	Total:	**3417.362**	**1899.105**	**1186.825**
Cyblue	i/o read&write	9.966552	131.6709	24.35653
	find RREF	8815.717	2461.659	2461.778
	find solution	4.041326	3.585266	65.55922
	Total:	**8829.725**	**2610.248**	**2551.694**

Figure 9.3 Matrix 2.

no effect on the results. We believe this is due to the nature of the matrices that we tested. Because the matrices were constructed at random they tended to be quite uniform from left to right. In other words, reordering did not result in moving the less-sparse columns to the right because there were no less-sparse columns to begin with. This stands in contrast to our crypto matrices. On those matrices the algorithm was unable to finish without reordering. By their nature the crypto matrices were not at all uniform.

From Figures 9.5, 9.6 and 9.7 we can see that lightning showed the best performance. On the other machines, increasing the number of threads sometimes worsened the running time. We believe that again, it is the nature of our test matrices that accounts for this

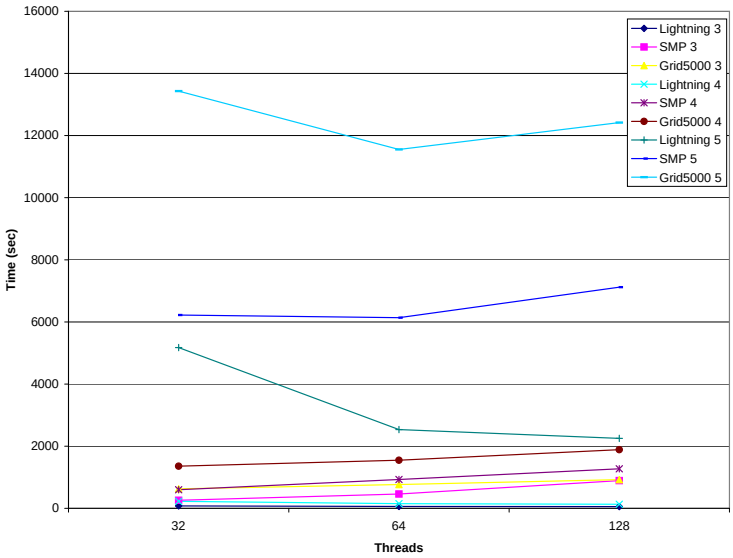


Figure 9.4 Matrix 3, 4 and 5.

strange behavior. These matrices were exceedingly sparse, approximately 4 non-zeros per row. The way our algorithm works, every row operation results in two MPI communication calls (`MPI_Allgather` and `MPI_Broadcast`). With such sparse rows, most of the running time was probably spent on communication. Since each platform uses different communicators, MPI calls can take longer on one than on others. We plan further investigation into the underlying causes of the performance differences and how to address them.

One thing we notice is that the I/O time is generally not significant. Typically we are talking about a matter of seconds. However we see that the I/O time for Matrix 3 with 128 threads on lightningSMP was 130 sec. This is quite out of scale compared to all of the other times. We conjecture that it is competition from other jobs running concurrently that slows down the I/O. Since we were running several of our test matrices simultaneously, we may have slowed ourselves down!

Finally, our study showed that the load balance function was never invoked. Again, we attribute this to the uniform nature of the test matrices. This is in contrast to the cryptography matrices, for which load balancing was essential.

		32	64	128
lightning	i/o read&write	5.147227	4.299652	5.286664
	find RREF	57.27072	44.7574	45.50847
	find solution	15.75639	6.24439	8.876194
	Total:	**78.17434**	**55.30144**	**59.67133**
SMP	i/o read&write	3.217348	7.014034	129.7484
	find RREF	242.7683	413.0013	602.9419
	find solution	14.34756	40.9309	158.2444
	Total:	**260.3332**	**460.9462**	**890.9347**
Grid5000	i/o read&write	3.297211	3.313362	7.468517
	find RREF	617.5841	752.8128	905.719
	find solution	7.923695	9.799342	9.191952
	Total:	**628.805**	**765.9255**	**922.3795**

Figure 9.5 Matrix 3.

		32	64	128
lightning	i/o read&write	13.95488	7.029839	6.725169
	find RREF	193.6696	129.2694	110.897
	find solution	17.09733	13.76376	14.07818
	Total:	**224.7219**	**150.063**	**131.7004**
SMP	i/o read&write	12.98728	8.062209	7.279615
	find RREF	552.0777	852.5517	1219.997
	find solution	32.79713	66.81489	44.95949
	Total:	**597.8621**	**927.4288**	**1272.236**
Grid5000	i/o read&write	8.176585	5.663151	5.262402
	find RREF	1335.039	1527.113	1864.243
	find solution	14.89192	16.4095	17.81099
	Total:	**1358.107**	**1549.186**	**1887.317**

Figure 9.6 Matrix 4.

9.5 Performance results of experimental platforms

The four charts below show how each platform handled the task of solving matrices of different sizes. We should mention that matrices 1 and 2 have higher density and solving those matrices can take more time than solving larger matrices with smaller density. We decided to divide matrices into two groups: small and large. The number of non-zero entries per row in each category is almost the same. We wanted our runs to complete in reasonable time. This is why the large matrices are less dense. The original cryptography matrices were more dense. It sometimes took a couple of days to get them to the row reduced echelon form.

Figure 9.8 shows the results from the lightning cluster. We can see that the solution time for each matrix decreases as the number of threads increases. That is, of course, what we expect

		32	64	128
lightning	i/o read&write	42.4997	22.52167	60.40848
	find RREF	5015.795	1862.399	1590.321
	find solution	115.8615	54.69781	71.12552
	Total:	**5174.157**	**1939.619**	**1721.855**
SMP	i/o read&write	16.93715	108.8164	31.09077
	find RREF	4733.855	4331.838	5343.936
	find solution	65.44293	393.4341	282.5045
	Total:	**4816.235**	**4834.088**	**5657.531**
Grid5000	i/o read&write	20.18628	78.78962	70.0332
	find RREF	13223.1	11443.33	12224.47
	find solution	187.3284	29.47163	122.4516
	Total:	**13430.62**	**11551.59**	**12416.96**

Figure 9.7 Matrix 5.

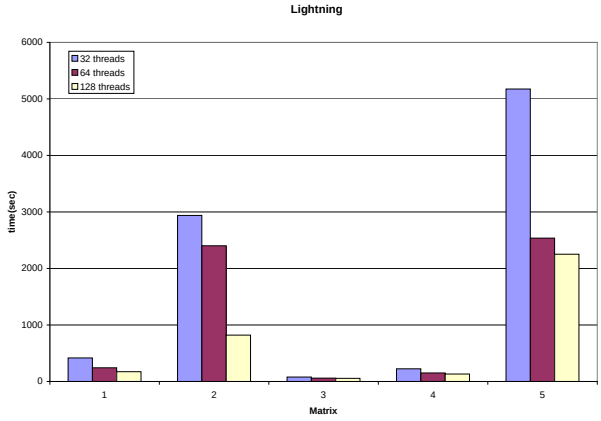


Figure 9.8 Lightning.

to happen. The chart does not show the running time for the other steps (I/O, assembling the solutions) because the times were insignificant compared to the solving step.

Figure 9.9 show the results for lightningSMP. The machine shows good results for smaller matrices. However, it is easy to see that for larger matrices the running time increases with the number of threads. This is obviously undesirable. As we explained earlier, we attribute this to the high proportion of time spent on communication as compared to calculation. The difference in performance compared to lightning is presumably due to the communication strategy on each machine.

Figure 9.10 shows the results from a supercomputer, Cyblue. The research shows that there is significant improvement between 32 and 64 threads and between 128 and 256 threads, but

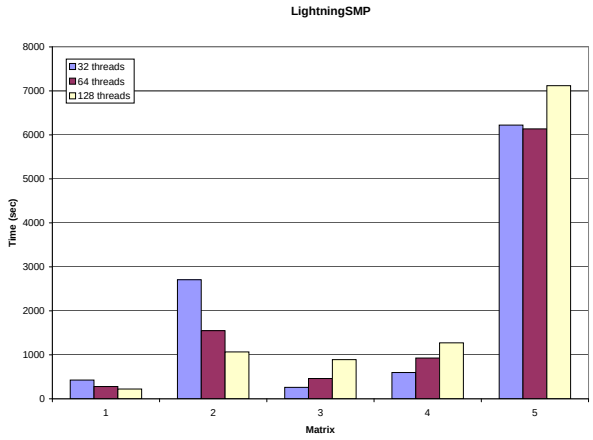


Figure 9.9 LightningSMP.

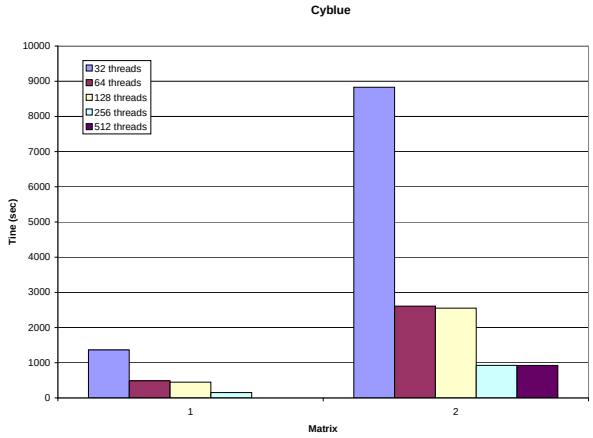


Figure 9.10 Cyblue.

little difference between 64 and 128 or between 256 and 512 threads. We were unable to conduct on Cyblue the same research we conducted on the other machines. Since our applications are memory extensive, the machine was running out of memory. This supercomputer architecture is not an appropriate platform for our purpose.

Figure 9.11 shows the results from the experimental platform Grid5000. The running time results were mostly much worse than the results from lightning and lightninSMP clusters. It is hard to explain the huge difference in running times for larger matrices. We know that different compilers were used. Our algorithm involves many loops. The main loop is on the pivot element and is done once for each unknown. For a matrix with $5,000,000$ unknowns it is

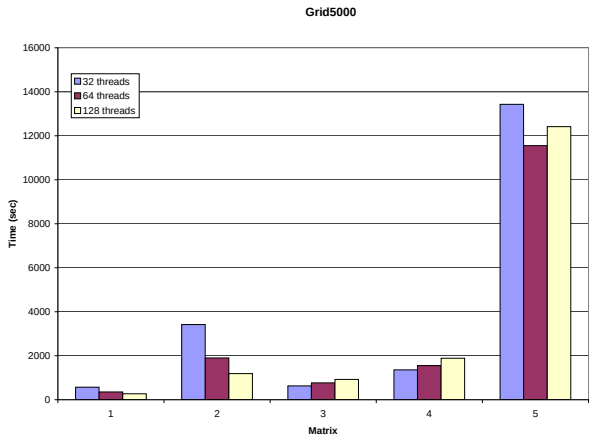


Figure 9.11 Grid5000.

a long loop. We also have four *if* statements inside "eliminate the 1 on pivot place" routines. A good optimization on loops and *if* statements can make a big difference. We plan to continue running our experiments with different optimization options. The result presented in this dissertation are for the runs with -O3 compiling option. The run without any optimizations doubles the time.

CHAPTER 10. Conclusions

We conduct the XL and XSL attacks on Baby Rijndael. Our study shows that the XL method works for one round Baby Rijndael. However, a four round Baby Rijndael can not be cracked using XL or XSL methods when only one block of plaintext and corresponding ciphertext are provided. The equations in the constructed systems are linearly dependent and number of possible solutions for these systems is larger than the number of possible keys for a brute-force attack. The XSL attack on four round Baby Rijndael with two blocks of plaintext and cipher text seems more promising. Unfortunately, we were unable to solve the system, since the matrix has grown rapidly. The authors of (4) also suggest that the attack might need more than one block of plaintext available.

In order to solve a large system of linear equations over the field $GF(2)$ that arises from this cryptography project, a new parallel software package called SPSOLVEMOD2 was created. This package is, to our knowledge, the first parallel implementation designed to find all solutions of a large, sparse system of linear equations over the field $GF(2)$. The package uses a novel input/output format, performs load-balancing when necessary, and includes an optional preprocessing step that reorders the columns of the matrix in an effort to reduce fill-in during the row-reduction step.

In Chapter 9 we have discussed our first performance tests of the software. The linear systems that we solved were large, very sparse and random. The uniform nature of the systems had an unexpected impact on the solution time. This is in marked contrast to cryptography systems we have solved.

We tested our software on several different computing platforms, with noticeably different results. Gaussian elimination is a memory-intensive enterprise, and an abundance of processors

will not compensate for a shortage of RAM. We have evidence that the communication design of a particular platform has a significant impact on performance.

There is still much work to be done in this area. The nature of the system to be solved and the communication overhead we experienced on very sparse systems should be investigated further. However we believe that SPSOLVERMOD2 has the potential to be a useful tool in several areas of discrete mathematics and cryptology. We hope that by improving the software we might be able to investigate more complex systems of linear equations rising from our Cryptography research. It should give more insight on whether the XSL attack can be efficiently used in order to crack AES.

BIBLIOGRAPHY

[1] *Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197, 2001.

[2] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Vicat-Blanc Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, B. Quetier and O. Richard, *Grid'5000: A Large Scale and Highly Reconfigurable Grid Experimental Testbed*, SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing CD, Seattle, Washington, USA: IEEE/ACM (2005), pp. 99–106.

[3] N. Courtois, A. Klimov, J. Patarin and A. Shamir, *Efficient Algorithms for solving Overdefined System of Multivariate Polynomial Equations*, Eurocrypt'2000, LNCS 1807, Springer-Verlag, pp. 392–407.

[4] N. T. Courtois and J. Pieprzyk, *Cryptanalysis of Block Cipher with Overdefined System of Equations*, Asiacrypt 2002, Volume 2501, Springer-Verlag, pp. 267–287.

[5] J. Daemen and V. Rijmen, *The Design of Rijndael*, Springer-Verlag (2002).

[6] A. S. Fraenkel and Y.Yesha, *Complexity of problems in games, graphs and algebraic equations*, Discrete Applied Math, Volume 1, no. 1-2 (1979), pp. 15–30.

[7] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun,J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham and T. S. Woodall, *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*, Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (2004), pp. 97–104.

[8] A. George and J. W. H. Liu, *The evolution of minimum degree ordering algorithm*, SIAM Review, vol. 31, no. 1 (1989), pp. 1–19.

[9] K. Hoffman and R. Kunze, *Linear Algebra*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J. (1971).

[10] http://bluegene.ece.iastate.edu, Cluster description (2010).

[11] http://crd.lbl.gov/ xiaoye/SuperLU/, Software description (2010).

[12] http://hpcgroup.public.iastate.edu/HPC/lightning/, Cluster description (2010).

[13] http://hpcgroup.public.iastate.edu/HPC/lightningsmp/, Cluster description (2010).

[14] http://www.netlib.org/scalapack/, Softare description (2010).

[15] A. Kipnis and A. Shamir, *Cryptanalysis of the HFE Public Key Cryptosystem by Relinearization*, Crypto99, LNCS 142,144, Springer-Verlag, pp.19–31.

[16] E. Kleiman and C. Bergman, *Parallel Reduction of mod 2 Sparse Matrices: Design and Performance Evaluation*, submitted to Supercomputing 2010.

[17] E. Kleiman, *The XL and XSL attacks on Baby Rijndael*, Master Thesis.

[18] B. A. LaMacchia and A. M. Odlyzko, *Solving large sparse linear systems over finite fields*, Advances in Cryptology: CRYPTO '90, Springer, Lect Notes Comput. Sci., 537, pp. 109–133.

[19] J. W. H. Liu, *Reordering sparse matrices for parallel elimination*, Parallel Computing, vol. 11 (1989), pp. 73–91.

[20] J. W. H. Liu and A. H. Sherman, *Comparative analysis of the Cuthill-Mckee and the reverse Cuthill-Mckee ordering algorithms for sparse matrices*, SIAM, Numer. Anal. 13, 2 (1976), pp. 198–213.

[21] C. S. R. Marthy, K. N. Balasubramanya Murthy and Srinivas Aluru, *New parallel algorithms for direct solution of linear equations*, John Wiley & Sons, INC. (2000).

[22] K. Nyberg, *Differentially uniform mapping for cryptography*, Advances in Cryptology, Proc. Eurocrypt'93, LNCS 765, T.Helleseth, Ed., Springer-Verlag (1994), pp. 55–64.

[23] A. M. Odlyzko, *Discrete logarithms in finite fields and their cryptographic significance*, T. Beth, N. Cot, I. Ingemarsson, eds., Lecture Notes in Computer Science 209, Springer-Verlag (1985).

[24] Y. Saad, *Iterative methods for sparse linear systems. Second Edition*, SIAM (2003).

[25] D. R. Stinson, *Cryptography: Theory and Practice. Second Edition,* Chapman and Hall/CRC (2002).

[26] D. H. Wiedemann, *Solving sparse linear systems over finite fields*, IEEE Trans. Information Theory, IT-32 (1986), pp. 54–62.