Algebraic Attack on Small Scale Variants of AES using Compressed Right Hand Sides

John - Petter Indrøy

Secure and Reliable Communications

University of Bergen

February 2018

Supervisor: Håvard Raddum

Abstract

The Advanced Encryption Standard is probably the most used symmetric encryption cipher in use today, which makes it particularly interesting for cryptanalysis. This thesis attacks small-scale variants of AES through a particular branch of algebraic cryptanalysis known as Compressed Right-Hand Sides. We see some success, as we are able to break for the first time three rounds of a 32-bit small-scale variant. We also make an interesting discovery, in that we get indications that some plaintext values result in easier-to-break small-scale instances.

Acknowledgment

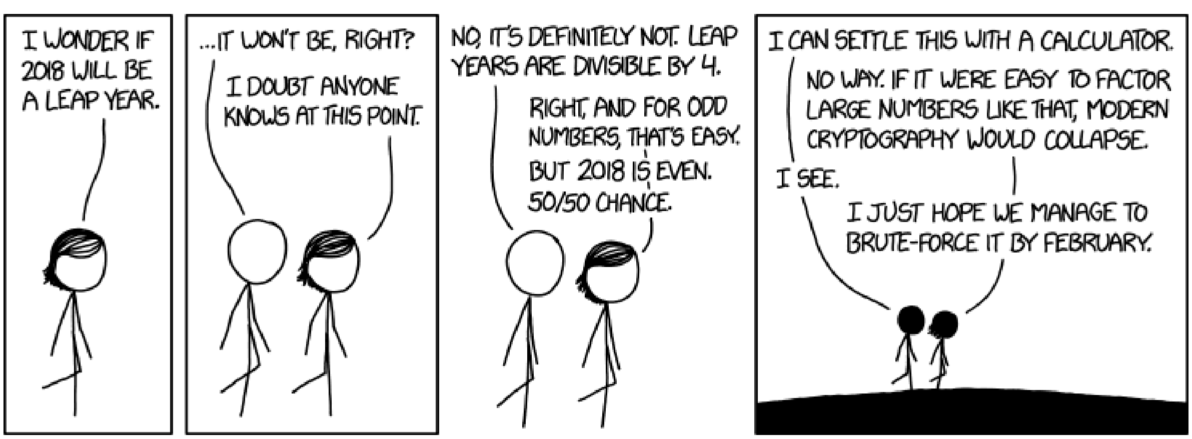
This work would not have been possible without the support and understanding of many people, of which I am deeply grateful.

My heartfelt thanks to my adviser, Dr. Raddum. I am thankful for his patience, guidance and continued support during this research.

To my parents, for relentless proofreading and help to conquer Excel, for a great upbringing and support through my whole life, for who you are, I am deeply thankful and appreciative.

To my remaining family and friends, you have shown great patience and understanding, especially during these last weeks. I look forward to seeing you again, and you have my thanks.

And to the most important and dearest person to me, my wife Helga. Without you, I would never have come as far. I am ever greateful.



"2018", by XKCD [8].

Contents

Chapter 1

Introduction

The internet has become enormously large and complex, with billions of everyday users. These users expect things to work, and they expect to use it without becoming subject to malicious intent. For instance, they expect that the right amount is drawn from their account when paying someone online, and that only themselves and their bank know how much they have on their account. There are numerous mechanisms in place attempting to ensure that using the internet is safe as possible. This thesis aims to take a closer look at one of those mechanisms, namely the Advanced Encryption Standard, or AES. To understand what the AES is, we need to explain what encryption is and how encryption is relevant to safe usage of the internet. Therefore, we also find it natural to talk about how we gauge the security of encryption, which will eventually lead us to what is known as cryptanalysis. I will then take the opportunity to give the problem statement in general terms. A more detailed problem statement comes at the start of chapter 5, as it is leans on background covered in chapters 2 til 4. We will then round of this chapter by giving an overview of the remainder of this thesis.

1.1 The Concept of Encryption

Encryption is the art of rendering readable text into something that looks like garble. When talking about digital messages, then ideally the garble should look no different than the random noise that naturally occurs when transferring data digitally. Hence we aim for the garbled message to look completely random. When designing modern ciphers there are two aspects related to this perceived randomness which we will consider after we have got some terminology in place.

Text that is readable to anyone is called *plaintext*. Plaintext that has been encrypted into a seemingly random string of letters (or bits, in case of computers), is called *ciphertext*. Creating a ciphertext from a plaintext is called to *encrypt* the plaintext. Making it back to readable form is called to *decrypt*. The steps taken to transform a plaintext to a ciphertext and back again is called an *encryption algorithm*, or a *cipher*. As we will see when we consider the application of encryption, we need one more component to make this useful, namely the *key*. A key is needed both to encrypt plaintext and to decrypt a ciphertext. No one who does not know the key should not be able to extract any information about the plaintext from the ciphertext. Keeping the key secret is therefore important. An illustration of the encryption process is given in Figure 1.1.

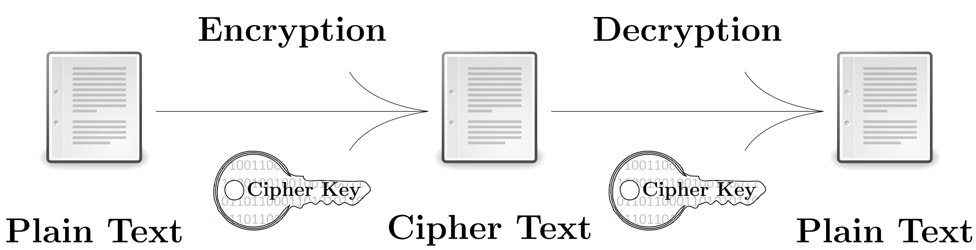


Figure 1.1: Encryption

The inclusion of keys ties in neatly with the randomness we want in the ciphertexts. Assuming the key is chosen at random, the randomness in the key should be enough to make the plaintext and the ciphertext seem completely unrelated. More generally, given two plaintexts encrypted under the same key, the resulting ciphertexts are supposed to yield no useful information at all about the two plaintexts. Even if only of bit, the smallest electronical building block of computers, is changed, the difference in the ciphertexts must look the same as if all the bits, or any other number of bits, were changed. More precisely, changing just one bit in one end (plaintext or ciphertext) should result statistically in the change of approximately half the bits in the other end.

There are two more principles important to modern-day ciphers. The first one is arguably the most important one, namely Kerckhoff's principle: If nothing but the key used is secret, the cipher should still be secure to use. This implies that even if some malicious third party knows every single detail about the cipher, not just how it works but also both the plaintext and the corresponding ciphertext under a key, but not the key itself, this third party should still not be able to somehow figure out the key in use.

The second principle is known by some as Schneier's Law, but, as Schneier himself points out [16], this principle outdates him. The principle states that *"anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."* [16]. What he means by this is that not being able to break your own cipher does not mean it is unbreakable. As trivial as that may seem, it still is important to bear in mind. For the modern day ciphers we use, there is no formal *proof* that they are unbreakable.

Because of these two principles, it is widely accepted as best practice to always publish a new cipher into the wild, so to speak, for others to scrutinize the cipher. If many clever people have tried hard to break a cipher, but failed, we can be relatively certain that no one can break it. This is also why this thesis is possible and relevant in the first place.

1.1.1 Symmetric vs. Asymmetric Encryption

As mentioned, encryption is only one of many security mechanisms in play. Its most noteworthy application is the end-to-end protection it offers messages sent over the internet. Imagine that Alice wants to talk to her online bank, Bob. If Alice were to send her message to Bob in plaintext, anyone along the way could read her message. Naturally, Alice would rather like that no other entity than Bob can read their communication. As most people do, she prefers her financial details to remain confidential, and she therefore decides to encrypt her message. She applies her chosen cipher combined with her secret key on her plaintext. Alice then sends the created ciphertext instead. This way no one that does not know the secret key can read the message. Since Bob also knows this secret key, he can decrypt and read Alice's message. Likewise, Bob can use the secret key to encrypt messages to Alice. This kind of encryption, where the same key is used to both encrypt and decrypt the message, is known as *symmetric encryption* and is illustrated in Figure 1.2.

A very useful consequence of using symmetric ciphers, is that it provides an indirect way for Alice to identify who she is talking to. If she trusts Bob not to share the key with anyone else, she can trust that it is Bob she is talking to. Since no one else but Alice and Bob have the key, no one else can encrypt a message that their secret key decrypts.

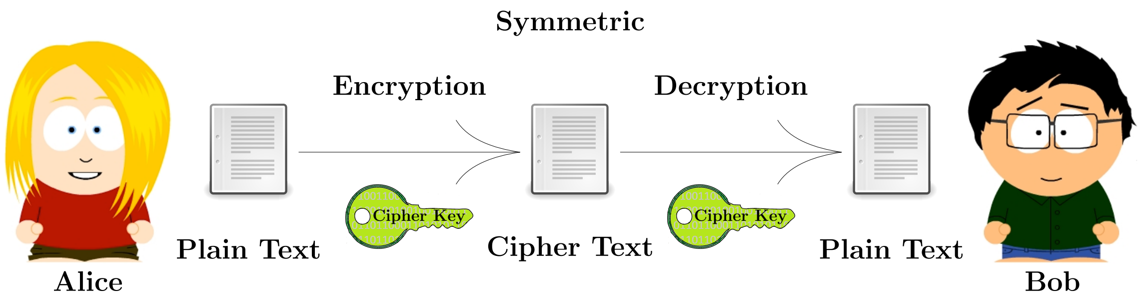


Figure 1.2: Symmetric Encryption.

This raises the question; how can Alice and Bob exchange the secret key in the first place? One way would be to send the key by some means in the “snail-mail". Fortunately, there exists a way to exchange keys online: By using *asymmetric cryptography*.

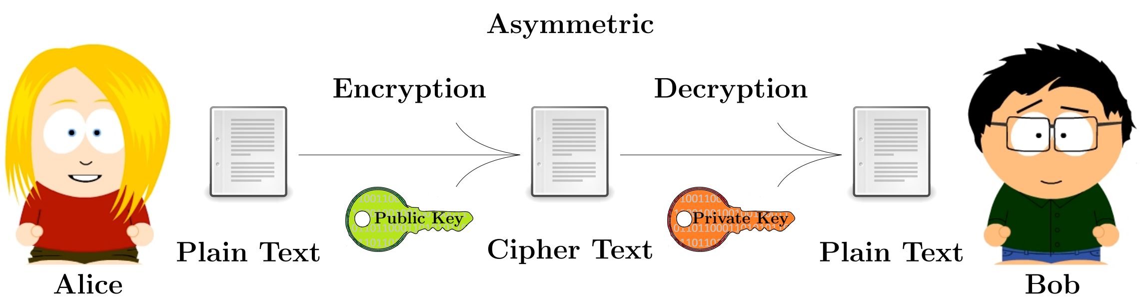


Figure 1.3: Asymmetric Encryption.

In asymmetric cryptography, see Figure 1.3, we use two keys instead of just one as in the symmetric case. One key, the *public key* is used to encrypt the message while the other key, the *private key* decrypts the message. In other words, the key that encrypted the message is not also capable of decrypting it!

As the names suggest, one key is shared publicly to anyone who wants it, while the other is kept utmost secret. If Alice wants to communicate with Bob, she can look up Bobs public key online and use that key to encrypt her message. She then sends the message to Bob, and if Bob has not shared/lost his private key, she can be confident that only Bob can decrypt the message using his private key. The drawback with asymmetric cryptography is that it is significantly slower than many symmetric cryptographic ciphers. Therefore, it is common to use asymmetric ciphers to exchange the symmetric key with the recipient, and then to switch to a symmetric cipher. Asymmetric ciphers are also useful to “prove” one’s identity, but that must stay a topic for another time. We will deal only with a symmetric cipher in this thesis, namely the Advanced

Encryption Standard.

1.2 Benchmark for the Security of Encryption

Since encryption is such an important part in staying safe while using the internet it is only natural to wonder about the strength of it. The first thing that we need to realize when talking about the security of ciphers is that no matter which one we choose, it may always be broken. This may sound odd at first, that we willingly use something we know may be compromised, and we even claim that is it safe to use. The explanation to that lies in the nature of the keys. A key is a string of bits of a length predetermined by the cipher in question. In other words, it is simply a long string of 0's and 1's, and anyone with the key can decrypt any ciphertext made under that key. This means that an attacker can attempt to decrypt any ciphertext by trying all possible variations of 0's and 1's of the specified length. Such an attempt is known as a *brute force* attack. A brute force attack will always be possible against any cipher that uses a secret key, in other words all ciphers in use today, but that does not mean brute force is viable. For example, the smallest key size used by AES is 128 bits long. Basic combinatorics tells us that since we have 128 places that each can hold either a 0 or a 1, we have 2^{128} possible keys. In other words, guessing a key at random gives a $1/2^{128}$ chance to succeed.

Even the fastest supercomputer alive today does not come close to brute-forcing 128 bit keys. The Sunway TaihuLigth supercomputer currently holds the title of fastest supercomputer [1], and can do ≈ 100 petaflops, or $\approx 2^{56}$ flops. Let us assume that we can test one key per flop. This is a simplification which allows for more keys to be tested at one time than realistic, yet let us increase our capabilities even further by assuming that we have 1000 such supercomputers. Then we can test a staggering 2^{66} keys per second! In terms of years, that is:

$$2^{66} \times 60 \times 60 \times 24 \times 365 = 2^{90} \text{ keys a year.} \tag{1.1}$$

Even accounting for the fact that we may expect to find a match after testing approximately half of the possible keys, we do not come close:

$$\frac{2^{128}/2}{2^{90}} = 2^{37} \approx 137 \text{ billion years is needed to find the key.} \tag{1.2}$$

Therefore, we define a cipher as secure if it is *computationally infeasible* to guess the key. Furthermore, we define a cipher as *broken* if there exists a method to find the key faster than by brute force. Notice that a cipher may still be regarded as secure even if it is broken, since it may still be computationally infeasible to find the key.

1.3 Cryptanalysis

We know that brute force is always possible, even though not practical. We would like to assure ourselves that brute force is the best attack we can do. In most cases we cannot know if there exists a better, more efficient way to find the secret key. The best we can do is to look for a better way. This is known as *cryptanalysis*, the science or art of breaking cryptosystems. Currently this is the best way of gauging and ensuring the security of cryptographic algorithms.

Because of the somewhat vague definition of cryptanalysis, we can divide these efforts into roughly three categories. It is important to note that opinions differ on whether the second and third category is included or excluded.

Classical cryptanalysis deals with attempting to recover the secret key from the associated ciphertext only, or from both the plaintext and ciphertext encrypted under some unknown key. This is usually done through various mathematical techniques and rigorous analysis of the algorithm in question. These techniques may be of a highly advanced level, or as simply as counting the frequency of letters. Brute force is an example of a technique in this category. Because it sets the bar for "worst-case" it also serves as a benchmark for how well the other techniques do. This is the category that everybody agrees on to be cryptanalysis.

Implementation attacks, or *side-channel attacks* is the second category. This one tries to obtain the key in use by exploiting weaknesses in how the cipher is implemented in a real-world application. This category is more debated since there are two kinds of exploitation possible: The one that looks for the key, and the one that looks for a way to bypass the key. Most cryptographers agree that the first one is regarded as cryptanalysis, while the second exploitation is more debated as to whether it should be considered cryptanalysis. To give an example, finding a key through the means of monitoring the power usage of a CPU during execution of a cipher is attempting to acquire the key, while exploiting race-condition to bypass an authentication process may be outside what many cryptographers consider to be cryptanalysis. Therefore, the group is somewhat debated.

The last group is *social engineering*. Bluntly said, this group encompasses all attempts to lure the victims into giving their keys to the attacker. A typical example of this is a phishing attack, where the victim is lured onto a website it believes belongs to credible company, when in reality it is the attacker's own website made to look like the credible company. If the victim attempts to log in with the credentials they use on the site they believe they are on, instead of actually logging in they give their credentials to the attacker. Since this way of obtaining the key, or equivalently, access to the system in question, is more of a bypass than an actual attack on the mathematics or implementation of the system, most cryptographers do not consider this category as part of cryptanalysis. However, in [10] they do.

Those who argue that it belongs to the term 'cryptanalysis' argues that the secret ingredient was acquired, and also that the system needs to take into account the human element. No matter what your opinion on this matter is, for a system to be secure overall, we need both strong ciphers and to make sure that successful implementation and social engineering attacks are as unlikely as possible.

When we use the term "cryptanalysis" in the remainder of this thesis, we think of it as "classical cryptanalysis" as defined above.

One last thing before we are ready to state our problem. As we will see in Chapter 4, attacking "normal" AES will take so much time that we cannot get any useful results from it. Therefore, in the second part of Chapter 3, we will consider small-scale variants of AES. In a nutshell, we vary several of the parameters set for the AES algorithm to create new, but similar, encryption algorithms that may actually be broken. It is believed that these small-scale variants retain much of the structure of "full" AES, and that any weaknesses found in a small-scale variant may give insight on the security of the full version. Using small-scale variants enables us to get useful data to analyze within a practical time frame. The details will be covered in Chapters 3 and 4.

1.4 Problem Statement for the Thesis

This thesis considers a particular branch of cryptanalysis known as *algebraic cryptanalysis*, applied to small-scale variants of the AES. We build on earlier work done in [3, 12, 11], and try to extend the results found there by attacking more AES variants with newer methods for algebraic attacks. The results show when we are successful in breaking small-scale versions of AES, and fill a small gap in our knowledge about the security of the AES.

We decided to put the full problem statement at the beginning of Chapter 5, as we feel that the full problem statement need more background covered.

1.5 Thesis Outline

Chapter 2 is intended to give a recap of important concepts relevant to the subsequent chapters: Abstract and linear algebra, Boolean functions, block ciphers, and cryptanalysis. Then we will move onto the Advanced Encryption Standard in Chapter 3, going into the details of the encryption algorithm. Here we will also cover small-scale AES, a common framework for the analysis of AES-like equation systems [3]. Chapter 4 covers the background and theory of the algebraic cryptanalysis branch we will use; Multiple Right-Hand Sides (MRHS) and Compressed Right-Hand Sides (CRHS). In here we also introduce three different solving strategies that utilizes CRHS. Next, Chapter 5 starts of by explaining the project setup and configurations. It then summarizes our results, before we discuss these findings. Lastly, in Chapter 6 we give some closing remarks and work for the future.

Chapter 2

Background

This chapter introduces the basic mathematics necessary to understand AES and the algebraic cryptanalysis of it that comes later in the thesis. Much of the content here is learned from the book "The design of Rijndael" by Daemen and Rijmen [13]. Furthermore, both the Boolean Functions and Block Cipher sections are inspired by the same book, though most, if not all, may be considered common knowledge in the field.

2.1 Abstract Algebra

The mathematical foundation of the Advanced Encryption Standard, as for many other cryptosystems, are based upon the field of abstract algebra. This section aims to give a recap of the most relevant concepts of abstract algebra as it pertains to this research, and is adapted from [13]. For a more comprehensive treatment of abstract algebra, consult an algebra book such as [4]

2.1.1 Group

In abstract algebra, groups are the basic construction on which more advanced mathematical constructs are built. It is therefore natural to begin by defining a group.

Definition 1. A group $< G, + >$ consists of a set G and an operation defined on its elements, here denoted by +:

$$+ : G \times G \to G : (a, b) \mapsto a + b,$$

fulfilling the following conditions:

- Closed: $\forall a, b \in G : a + b \in G$
- Associative: $\forall a, b, c \in G : (a + b) + c = a + (b + c)$
- Neutral element: $\exists 0 \in G,$ such that $\forall a \in G : a + 0 = a$
- Inverse elements: $\forall a \in G, \exists b \in G$ such that $a + b = 0$

Another possible condition the operation may satisfy is *commutativity*:

$$\textit{Commutative:}\ \forall a, b \in G : a + b = b + a$$

If the operation also is commutative, we call the group an *Abelian group*.

Example 1. There are two well known examples of Abelian groups that we use every day: the first is the set of integers under addition: $< \mathbb{Z}, + >$. The second is the structure $< \mathbb{Z}_{24}, + >$, which is used in 24 hour watches. It contains the integer numbers 0-23. The operation is addition modulo 24. This last example can be generalized to the structure $< \mathbb{Z}_n, + >$ which contains the set of integers from 0 to $n - 1$, with addition modulo n being its operation.

Since the set of integers under addition is the best-known example of a group, it is commonplace to use "+" to denote an arbitrary group operation. Also, "+" is often referred to as "addition". We will adhere to this practice in this thesis both when talking about an arbitrary group operation as well as talking about integer addition. The context should make it clear what operation the symbol is referring to.

2.1.2 Ring

The next structure to define is the ring. A ring is essentially an Abelian group that has been "expanded" with a second operation. The second operation needs to have a neutral element, associativity and closedness, but it needs not have inverses. Therefore, the set under the second operation only, needs not be a group by itself.

Definition 2. A ring $< R, +, \times >$ consists of a set R with two operations defined on its elements, here denoted by $+$ and $\times$. In order to qualify as a ring, the operations have to fulfill the following conditions:

- The structure $< R, + >$ is an Abelian group
- The operation $\times$ is closed, and associative over R. There is a neutral element for $\times$ in R
- The two operations $+$ and $\times$ are related by the law of distributivity: $\forall a, b, c \in R : (a+b) \times c = (a \times c) + (b \times c)$.

The operator $\times$ is often referred to as "multiplication", and its neutral element is usually denoted by 1. If $\times$ is commutative, the ring $< R, +, \times >$ is called a commutative ring.

Example 2. Example: If we include multiplication in the set of integers under addition from the previous example, we get the ring $< \mathbb{Z}, +, \times >$, the set of integers under addition and multiplication. This ring is commutative. Another well known ring is the set of matrices over $\mathbb{Z}$ with n rows and n columns under "matrix addition" and "matrix multiplication". This ring is not commutative for n larger than 1.

2.1.3 Field

The next structure, the field, will expand the concept of a ring. Simply said, a field is a commutative ring that also has inverse elements with respect to multiplication.

Definition 3. A structure $< F, +, \times >$ is a field if the following two conditions are satisfied:

- $< F, +, \times >$ is a commutative ring.
- For all elements of F, there is an inverse element in F with respect to the operation $\times$, except for the element 0, the neutral element of $< F, + >$.

A field can be thought of as a set that is an Abelian group both under addition alone and under multiplication alone, except for 0. More formally, a structure $< F, +, \times >$ is a field if both $< F, + >$ and $< F\backslash\{0\}, \times >$ are Abelian groups and the law of distributivity applies. The neutral element of $< F\backslash\{0\}, \times >$ is known as the unit element of the field.

Example 3. The set of real numbers under addition and multiplication is the best-known example of a field. When a set is a field, it is possible to do addition, subtraction, multiplication and division without leaving the set. Subtraction is done by adding inverses: $a–b = a + (-b)$, where $-b$ is the additive inverse of b. Division uses the multiplicative inverses: $a/b = a \times b^{-1}$, where b^{-1} is the inverse of b with respect to multiplication.

2.1.4 Finite Fields

A finite field is a field with a finite number of elements. The number of elements in the set of the finite field is known as the order of the field. There can only exist finite fields for which the order is a prime power. More formally, there can only exists fields of order m if and only if $m = p^n$ for some integer n and p being a prime integer. This has to do with the need for inverses for both operations in the field. It is also worth noting that p is known as the characteristic of the finite field. An important property of finite fields is the fact that fields of the same order are *isomorphic*: Even though the elements of two fields of the same order may differ in their representation, their underlying algebraic structure is exactly the same.

Definition 4. Two finite fields F and F' are isomorphic if there exists a one-to-one function φ mapping F onto F' and the following conditions are satisfied:

- $\varphi(x + y) = \varphi(x) + \varphi(y), \forall x, y \in F$
- $\varphi(x \times y) = \varphi(x) \times \varphi(y), \forall x, y \in F.$

This means that for each prime power there exists exactly one finite field, denoted $GF(p^n)$. The perhaps easiest form of finite fields to grasp are the ones where $n = 1$. When this is the case, the finite field has order p, and due to isomorphism, the finite field can be represented by $< \mathbb{Z}_p, +, \times >$.

When the order is not prime, i.e. $n > 1$, things are a bit more complicated. The operations can no longer be modulo p, nor will they be modulo p^n. Instead we will represent $GF(p^n)$ as polynomials over $GF(p)$ of degree n. This is not the only way to represent $GF(p^n)$ with $n > 1$, but it is the one we will use in this thesis. The reason for this is that these polynomials in $GF(2^8)$ can easily be represented using Boolean vectors, which can conveniently be stored as 8-bit values, or bytes. This is opportune for us, since we will only be working with fields of characteristic 2, with $n \in \{1,4,8\}$. Table (2.1) gives $GF(2^4)$ as numbering the elements using hexadecimal notation and the corresponding 4 bit Boolean vector.

Hexadecimal	Boolean vector
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Table 2.1: Table of the elements of $GF(2^4)$. Use equation (2.1) to go from Boolean vector form to the corresponding polynomial.

2.1.5 Polynomials over a Field

A polynomial is a sum of a finite number of terms, where each term is a constant multiplied with one or more variables to the power of a positive integer exponent. A polynomial b over a field F is an expression of the form

$$b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + ... + b_1 x + b_0,$$

where the $b_i \in F$ are known as the *coefficients*. There is no need to evaluate the polynomials in this thesis, and we will therefore treat them as abstract elements only. The *degree* of a polynomial is the largest exponent in the polynomial which have a non-zero coefficient.

The set of polynomials over a field F is denoted $F[x]$. A compressed, efficient way of writing polynomials is to store only the coefficients as an ordered string. Since we will use polynomials with coefficients from $GF(2)$ in this thesis, the coefficients may only be 0 or 1. This enables us to store polynomials up to degree 8 in a single byte:

$$b_7b_6b_5b_4b_3b_2b_1b_0 \mapsto b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0. \tag{2.1}$$

Bytes are often written in hexadecimal notation.

2.1.6 Operations on Polynomials

Addition of polynomials consists of summing the coefficients of equal powers of x, where the summing of the coefficients occurs in the underlying field F. The neutral element for addition is the polynomial in which all coefficients are equal to zero. The additive inverse of a polynomial is easily made by replacing each coefficient by its additive inverse element in F. For the polynomial representation of the elements in $GF(2^n)$, each polynomial will be its own inverse under addition.

Definition 5. A polynomial $d(x)$ is *irreducible* over the field $GF(p)$ if and only if there exist no two polynomials $a(x)$ and $b(x)$ with coefficients in $GF(p)$ such that $d(x) = a(x) \times b(x)$, where both $a(x)$ and $b(x)$ are of degree > 0.

Definition 6. The multiplication of two polynomials $a(x)$ and $b(x)$ is defined as the algebraic product of the polynomials modulo an irreducible polynomial $m(x)$:

$$c(x) = a(x) \cdot b(x) \Leftrightarrow c(x) \equiv a(x) \times b(x) \pmod{m(x)}.$$

This makes the multiplication operation closed.

With respect to addition of polynomials, multiplication of polynomials is associative, commutative and distributive. The neutral element is the polynomial of degree 0 and with coefficient of x^0 equal to 1. In order to find the inverse for the multiplication, the Extended Euclidean Algorithm may be utilized (see e.g. [7, p. 81]).

2.1.7 Some Observations on Finite Fields with Characteristic 2

- Elements of finite fields with characteristic 2 may be represented as binary polynomials. This makes them easy to store and process digitally.
- Multiplication by x is fast when byte representation is used, as it is the same as a left-shift of the bits, followed by an addition of the chosen reduction polynomial if the highest coefficient is 1.

2.2 Linear Algebra

As linear algebra is one of the main pillar of MRHS and CRHS, this section will briefly cover some core aspects of it. This section is based upon [6]. For a comprehensive treatment, see [6] or some other linear algebra textbook.

A *linear equation* in the variables $x_1, \ldots, x_n$ is an equation that can be written in the form

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = b \tag{2.2}$$

where $a_1, \ldots, a_n$ are called the *coefficients*. A *system of linear equations* (or a *linear system*) is a collection of one or more linear equations involving the same variables.

Example 4.

$$\begin{aligned} x_1 + x_2 + x_3 &= 0 \\ x_1 + x_3 &= 1 \end{aligned}$$

The coefficients of a system of linear equations may be written as a matrix, in what is known as the *coefficient matrix.*

Example 5. The coefficient matrix of Example 4: $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$

The *size* of a matrix is the number of rows and columns that comprises it, denoted $m \times n$. A matrix with only one column is called a *vector*. If $\mathbf{b}$ is included in the coefficient matrix of Example 4, we have the *augmented matrix*.

Example 6. The augmented matrix of Example 4: $\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$

Reducing the augmented matrix in Example 6 into *echelon form* (see [6, ch 1.2]) quickly tells us if there exists a solution to Example 4. If there exists no row in the reduced augmented matrix on the form $\begin{bmatrix} 0 & \dots & 0 & 1 \end{bmatrix}$ then there exists at least one solution. Otherwise we have no solution. Further reducing the matrix into *reduced echelon form* makes it easy to tell if the solution is unique. If there are no *free variables* the solution is unique. Otherwise we have more than one solution, depending on the field we are in. For $\mathbb{R}$ we have infinitely many solutions, while for $GF(2)$, which is the one we will operate in, we have 2^k solutions for k free variables.

Given vectors $\mathbf{v}_1, \dots, \mathbf{v}_p$ in $GF(2)^n$ and given scalars $c_1, \dots, c_p$, the vector $\mathbf{y}$ defined by

$$\mathbf{y} = c_1\mathbf{v}_1 + \dots + c_p\mathbf{v}_p \tag{2.3}$$

is called a *linear combination* of $\mathbf{v}_1, \dots, \mathbf{v}_p$ with *weights* $c_1, \dots, c_p$.

The matrix equation $A\mathbf{x} = \mathbf{b}$ is the linear combination of the columns of A using the corresponding entries in $\mathbf{x}$ as weights, that is

$$A\mathbf{x} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = x_1\mathbf{a_1} + x_2\mathbf{a_2} + \dots + x_n\mathbf{a_n}$$

Example 7. $\mathrm{A}\mathbf{x} = \mathbf{b}$ form of Example 4: $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

A n-vector $\mathbf{x}$ is said to be a solution if it satisfies $\mathrm{A}\mathbf{x} = \mathbf{b}$, meaning that $\mathbf{b}$ is a linear combination of the columns of A using the entries of $\mathbf{x}$ as weights. A system is said to be *consistent* if it contains no rows on the form $\begin{bmatrix} 0 & \dots & 0 & 1 \end{bmatrix}$ when in echelon form. In other words, when there is at least one solution to the system. Otherwise it is said to be *inconsistent*.

A set $\{\mathbf{v}_1, \dots, \mathbf{v}_p\}$ of two or more rows of A is said to be *linearly dependent* if there exists weights $c_1, \dots, c_p$ such that

$$\mathbf{0} = c_1\mathbf{v}_1 + \cdots + c_p\mathbf{v}_p,$$

where not all c_i are 0. If there is no combination that forms the zero row, $\{\mathbf{v}_1, \dots, \mathbf{v}_p\}$ are said to be *linearly independent* of each other.

2.3 Boolean Functions

2.3.1 Bits and Boolean Vectors

The smallest finite field, $GF(2)$ has only two elements, 0 and 1. These elements are known as bits, or Boolean variables, depending on context. The two operations of this finite field, addition and multiplication correspond to the logical operations of XOR and AND respectively. XOR is a binary function that returns 1 if and only if the two input values differ, see Table (2.2). AND is a binary function that on the other hand returns 1 if and only if both input values are 1, see Table (2.3).

XOR	0	1
0	0	1
1	1	0

Table 2.2: Table for XOR of two bits.

AND	0	1
0	0	0
1	0	1

Table 2.3: Table for AND of two bits.

A vector whose coordinates are bits is called a Boolean vector. Often a Boolean vector is represented as a binary string of equal length to the vector. One can XOR or AND two Boolean vectors of equal size by XORing/ANDing the corresponding bits from each vector, called bitwise XOR and bitwise AND.

2.3.2 Function, Transformation and Permutation

A Boolean function $\mathbf{b} = \varphi(\mathbf{a})$ is a function that maps a Boolean vector to another Boolean vector.

$$\varphi : GF\left(2^n\right) \to GF\left(2^m\right) : \mathbf{a} \mapsto \mathbf{b} = \varphi\left(\mathbf{a}\right)$$

where **a** is called the input vector and **b** is called the output vector. If the output vector **b** has only one bit, that is $m = 1$, then φ is known as a Boolean function. When the input vector **a** has the same length as the output vector **b**, or $n = m$, φ is known as a *Boolean transformation*. A Boolean transformation may be viewed as a function that operates on a *state*. If the Boolean transformation also is one-to-one and onto, which makes it invertible, then we call φ a Boolean permutation. Onto means that every possible output vector is mapped to by some input vector to φ. One-to-one means that different input vectors always maps to different output vectors. Summarized, a Boolean permutation is a Boolean function that is invertible, and that has input vectors **a** of same length as its output vectors **b**.

2.3.3 Partition Bundles

When dealing with sets of binary variables, it is often useful to partition them into disjoint subsets known as bundles. This allows us to express functions in terms of these bundles instead of in terms of each individual bit. We will deal only with ordered sets, which has the effect that the bits within the bundles also will be ordered, and that the bundles among themselves, at least initially, will be ordered. We normally use indexes to keep track of the order, and the index scheme in use will be explained when needed.

By bundling together bits one can easily represent extensions of $GF(2)$. For instance, a bundle of 4 bits can be thought of as an element in $GF(2^4)$, with indexing starting at 0 and rightmost. Thus it corresponds with the indexing convention used in polynomials over a field, like in (2.1). Table (2.4) shows a bundle of four bits, and its corresponding polynomial in $GF(2^4)$.

Bundle	Polynomial
0101	$0x^3 + x^2 + 0x + 1$

Table 2.4: A 4-bit bundle and its corresponding polynomial.

2.3.4 Transposition and Bundle Transposition

A *transposition* $\mathbf{b} = \pi\mathbf{a}$ is a function that changes the order of an ordered set, without changing the values of the elements.

$$b_i = a_{p(i)},$$

where i is an index and $p(i)$ is a permutation of the indices. When the set is a set of bundles, this means that a permutation of the bundles will be executed, but the order of the internal bits of each bundle will remain the same. So even if the bundle order is changed, the values stay the same. This is known as a *bundle transposition*, and Figure (2.1) gives an example.

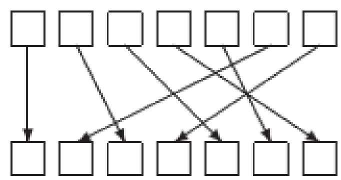


Figure 2.1: Example of a bundle transposition. From [13, p. 21].

2.3.5 Bricklayer Function

A similar, yet fundamentally different Boolean function to the bundle permutation, is the *bricklayer function*. The bricklayer function also works on smaller partitions of a set, but unlike the bundle permutations, it may, and usually do, change the values of the bits of its bundles. One may view it as a Boolean function that may be decomposed into a number of Boolean functions, each of whom operate in parallel on a partition. Note that these decomposed functions may be different from one another.

These decomposed functions are known as S-boxes when the function is non-linear, and D-boxes when they are linear. S stands for substitution while D stands for diffusion. When the input vector is of the same length as its output vector, we call the overall function for a bricklayer transformation. If the partitions/bundles within the input vector $\mathbf{a}$ and $\mathbf{b}$ are denoted by a_i and b_i respectively, this can be represented as $b_i = \varphi_i(a_i)$. It is worth noting that the parallel operations of the S-/D- boxes are independent from each other.

The non-linear step of the AES-candidate Serpent is an example of a bricklayer function. As are all AES' Boolean transformations, as we will see in the next chapter. If the S- / D- boxes of the bricklayer transformation are all invertible, the bricklayer transformation is also invertible, and thus known as a bricklayer permutation.

2.3.6 Iterative Boolean Transformation

One may apply Boolean transformations on a Boolean vector, one after another, creating a sequence of Boolean transformations known as an iterative Boolean transformation. Figure 2.2 shows the form of an iterative transformation, in where $\rho^{(i)}$ represents the individual transformations.

$$\beta = \rho^{(r)} \circ \ldots \circ \rho^{(2)} \circ \rho^{(1)}(a_1)$$

The value of $\rho^{(i)} \circ \ldots \circ \rho^{(1)}(a_1)$ for $1 < i < r$ is known as an intermediate state. If all the intermediate functions are Boolean permutations, the whole function is an iterative Boolean permutation, and is thus invertible.

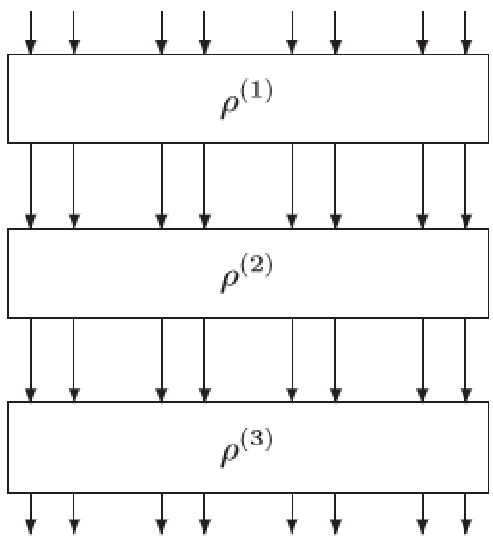


Figure 2.2: Illustration of an iterative Boolean transformation. From [13, p. 23].

2.4 Block Ciphers

A block cipher is a permutation that transforms plaintext blocks of a fixed length n_b to ciphertext blocks of the same length, under the influence of a cipher key k. One may view a block cipher as a set of operations that works on fixed length vectors. The key vector may be of a different length n_k.

For a fixed plaintext vector and a key vector of size n_k there are 2^{n_k} possible permutations for the block cipher. The act of transforming an input vector, or plaintext block, into an output vector, or ciphertext block, under the influence of the key k, is known as *encrypting* the plaintext under k. Transforming the ciphertext back into the plaintext using the key k, is known as *decrypting* the ciphertext under k.

The specification of the block cipher gives the *encryption algorithm*. The encryption algorithm specifies the operations to be used, and the sequence in which they will be applied to the plaintext in order to obtain the ciphertext. In this thesis we will only be dealing with plaintexts, keys and ciphertexts represented as Boolean vectors. This means that the only operations we will be dealing with are Boolean functions. Since encrypting a plaintext without the ability to decrypt it again is of little use to us, all Boolean functions will be Boolean permutations.

2.4.1 Key-Iterated Block Ciphers

According to [13], AES belongs to a class of block ciphers known as *key-iterated block ciphers*. In a key-iterated block cipher, the cipher is defined as the alternating application of round-transformations and key additions. One application of the round-transformation, or the key-independent Boolean transformation, and one key addition is one *round* of the cipher. It is normal to have a key addition step before the first round as well. The keys for each round are usually specified in a part of the encryption algorithm known as the *key schedule*. How the key schedule and round transformations are designed vary from block cipher to block cipher.

Key-iterated block ciphers' key addition step is simply to XOR in the round key. Furthermore, each round transformation, with the possible exception of the first or last round, need be the same. This makes for efficient implementation in both hardware and software. Key-iterated block ciphers belong to the class of key-alternating block ciphers. Figure 2.3 illustrates two rounds of a key-alternating block cipher.

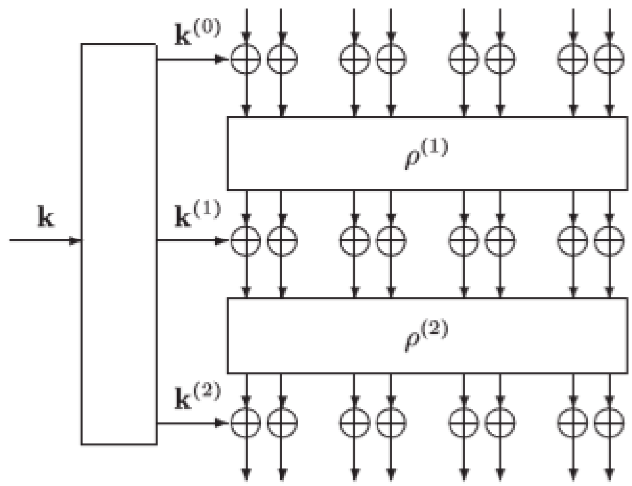


Figure 2.3: Key-alternating block cipher with two rounds. From [13, p. 26]

2.5 Cryptanalysis

Analysing ciphers to assess their strength is known as cryptanalysis. As already noted in Section 1.3 we only consider classical cryptanalysis in this thesis, i.e., only studying the abstract description of the cipher in question and not taking any particular use or implementation into account. There are some known standard techniques for doing cryptanalysis of block ciphers. The most well known (modern) cryptanalytic methods are called *differential* and *linear* cryptanalysis.

In differential cryptanalysis one considers two plaintexts at the time, usually with some small, known, difference between them. The crucial observation is that adding the same round key onto the two plaintexts will not change this difference! So when only considering differences of two cipher blocks, key additions behave like the identity mapping. Linear operations in the cipher change the difference of a cipher block, but in a known way. The only operations in a cipher that can change a difference in an unpredictable way are the non-linear ones, such as S-boxes. In a differential attack the attacker tries to predict what the difference of the two cipher blocks will be at some point in the encryption operation. If the attacker knows that a difference has a relatively high probability of occurring at some particular point close to the ciphertext, he can use this information to find what the value of at least parts of the last round key(s) must be. This is usually enough to break the cipher.

In linear cryptanalysis the attacker studies linear combinations of bits from the cipher block as it progresses through the cipher. Starting with a known plaintext, the attacker knows what the sum of some of its bits will be. After adding the first round key, the attacker knows that the sum of the same bits in the cipher block will have the same value if the corresponding bits in the round key sum to 0, and will be flipped otherwise. The crucial thing is that repeating this for many plaintexts, the attacker knows that the 0/1-distribution of the particular linear combination will be the same, or flipped, after adding key material. Applying linear transformations on the cipher block does not change this, the attacker still knows how skewed the 0/1-distribution is for some linear combinations at the output of a linear transformation. Again, the only component that defends against linear cryptanalysis are the non-linear ones, i.e. S-boxes. Different S-boxes gives better or worse protection, and linear cryptanalysis is most famous for being the best attack on the Data Encryption Standard (DES) that was the predecessor of AES. The S-boxes in DES do not give optimal protection against linear cryptanalysis.

The topic of the rest of this thesis is *algebraic* cryptanalysis. In algebraic cryptanalysis the attacker treats the unkown bits of the key as variables, and models the whole encryption algorithm as an equation system using the knowledge of one plaintext/ciphertext pair. The question of breaking the cipher then becomes a question of solving the equation system. In order to keep the equations of a manageable size the attacker normally needs to introduce more variables that represent the bits of the cipher block at certain points in the encryption process.Therefor the total number of variables in the system is usually quite a bit larger than just the size of the user-selected key. If all operations in a cipher are linear, the equation system describing the cipher would also be linear and hence very easy to solve. So for algebraic cryptanalysis as well, it is the non-linear components of the cipher that gives protection against this attack method.

Chapter 3

AES and Small-Scale Variants

On November 26th. 2001, the National Institute of Standards and Technology (NIST) published the Advanced Encryption Standard [9]. This was following a four year long process, where 15 candidates had been evaluated and dwindled down to just one. The global cryptographic community had been invited to analyse and to try to find weaknesses in the candidates, and after a thorough process, Rijndael was selected as the new standard [13].

Rijndael and AES as specified in [9] are not quite the same, as Rijndael has more flexibility to block sizes than what was required for AES. This chapter will therefore concentrate on AES as it is probably the most widely uesd symmetric cipher today. The first section will explain how AES transforms a plaintext into ciphertext and back again. As full AES is beyond what we are currently able to attack using the techniques in the next chapter, we spend the next section on small-scale versions of AES. Small-scale AES is a common framework for the analysis of AES-like equation systems [3]. This will allow us to attack smaller AES-like ciphers and see how small they must be for attacks to actually succeed on a normal computer. Any weakness found in any small-scale AES version would however need to be verified for the full AES.

3.1 Overview

The Advanced Encryption Standard falls into the key-iterated block cipher category, as it has N_r number of rounds consisting of the application of three Boolean permutations on the state followed by XORing the round key and the state. For the ease of use, and since it has no practical implications, we will include the addition of the round key, also known as the subkey, in the round transformation, even though this is inconsistent with the definition of key-alternating block ciphers from Section 2.4.1.

The AES encrypts blocks of 128 bits of plaintext into blocks of 128 bits ciphertext, and back. Since any message larger than 128 bits can be broken into bundles of 128 bits and encrypted/decrypted in parallel, AES may also be considered a bricklayer permutation.

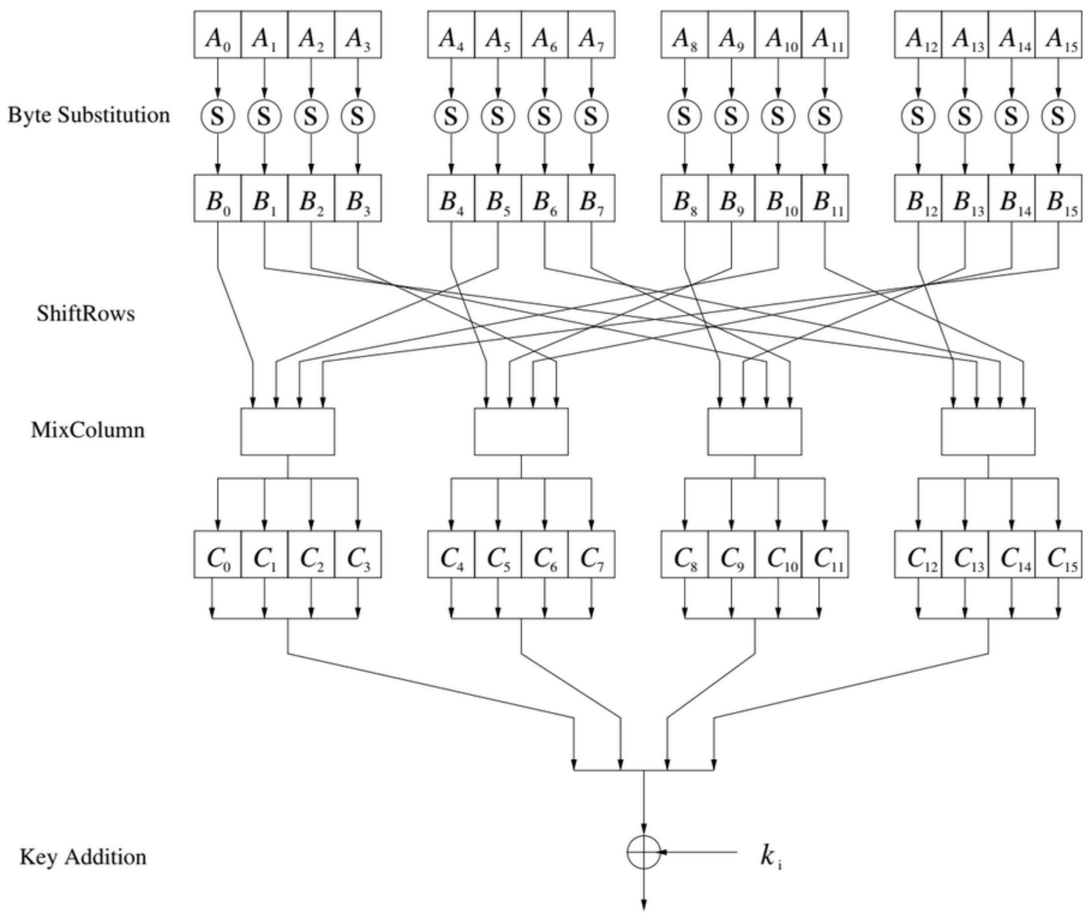


Figure 3.1: Graphical overview of rounds 1 to $N_r - 1$ of AES. From [10, p. 100].

Depending on the key size N_k, AES will have 10, 12 or 14 rounds. The N_r–1 first rounds all starts with the state going through the nonlinear SubBytes before going through a transposition of its bytes in ShiftRows, and then dependencies are created between the bytes in MixColumn. Finally, the round key is XORed onto the intermediate state. The step of XORing in the round key is named AddRoundKey. The last round follows the same pattern, except that Mix Columns is omitted. Lastly, before the first round and in what we have chosen to call the “pre-round”, the initial round key is XORed with the plaintext, creating the first intermediate state.

All the round keys are derived in the Key Schedule, an algorithm designed for the expansion of the original key into $N_r + 1$ round keys. The legal key sizes for AES are only three; 128 bits, 192 bits and 256 bits, all divisible by 32. The key size determines the number of rounds AES will utilize: 10, 12 and 14 rounds, respectively. The key schedule may have fewer rounds itself, though it will always produce precisely the needed number of subkeys. The concatenation of all the subkeys is called the expanded key.

```
AES(plaintext, key)
{
        KeySchedule(key)
        AddRoundKey(plaintext, expandedKey[0])

        for (i = 1; 1 < Nr ; i++) {
                Round(state, expandedKey[i]
        }
        FinalRound(state, expandedKey[Nr]

        return ciphertext
}
```

Figure 3.2: Pseudo code: high level overview of the AES. From [10].

```
Round(state, expandedKey[i])
{
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, expandedKey[i])
}
```

```
FinalRound(state, expandedKey[Nr])
{
        SubBytes(state)
        ShiftRows(state)
        AddRoundKey(state, expandedKey[Nr])
}
```

Figure 3.3: Pseudo code: Round and FinalRound. From [10].

3.2 Math in AES

Math in AES is done in the finite field $GF(2^8)$. All elements may be represented as integers, hexadecimal, binary or as polynomials. We will mostly use binary strings or polynomials. The primitive polynomial that defines the AES instance of $GF(2^8)$ is $x^8 + x^4 + x^3 + x + 1$. We quickly remind that multiplication by x, or 00000010 is the same as a left shift in binary, where an "overflow" results in an addition of the binary representative of the primitive polynomial: 00011011. Multiplication by $x+1$, or 00000011, is equal to multiplication by x followed by an addition of the original element itself. This may be done efficiently in both software and hardware.

3.3 Indexing in AES

AES is known as a byte oriented block cipher. This means that the main size of the bundles in AES are 8 bits large, or exactly one byte. All other bundle sizes are multiples of the byte. This is also true for the state, which is 128 bits large, or in terms of the bundles, 16 bytes.

When dealing with the state in AES' Boolean functions, the state will always be arranged in a four by four matrix. The indexing convention used to enumerate these bundles starts at the top-left bundle, naming it 0. Then it follows the column downwards, incrementing by one as it goes. Upon reaching the bottom of one column, it will proceed to the next column to its right, continuing the incrementation where it left off, until it reaches the bottom of the fourth column. See Figure (3.4) for an illustration. Note that this is contrary to what we are used to when reading, where we go top right and finishing the row before moving downwards.

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

Figure 3.4: AES state indexing scheme.

It should be specified that whenever an element of $GF(2^8)$ is written in binary and as a column, the enumeration will always start at b_0 at the topmost bit. When written as a row, we start enumeration from the rightmost bit towards the left. Again we start indexing with 0. The bit in position 0 is considered the *least significant bit*, bearing the same implications as the least significant digit in a ordinary number.

When enumerating the AES rounds, we will count the "pre-round", where only AddRoundKey is performed, as round 0, even though it is technically not a round. The first full round will be designated round 1, and so forth. The final round is indexed by N_r.

The fourth indexing scheme is that of the subkeys. The first 128 bit subkey is the one who will be used in the "pre-round", and will be designated subkey 0 or round key 0. Then follows the normal incrementation, where the last subkey is subkey N_r. This ends up giving $N_r + 1$ subkeys in total.

3.4 Round Operations

3.4.1 SubBytes

The *SubBytes* permutation is a bricklayer permutation that applies 16 parallel S-boxes on the input vector. Each s-box Sb takes a byte as input and then substitutes it with a predefined byte, hence the name S(ubstitution) – box. This permutation is the only non-linear permutation in AES. The construction of this mapping is a two-fold process, based upon the strong algebraic properties that $GF(2^8)$ offer. First step is mapping the bytes, regarded as elements in $GF(2^8)$, to their inverse in $GF(2^8)$ under the irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. Since 00 has no multiplicative inverse, it is mapped to itself. In the next step each byte is regarded as a vector over $GF(2)$ and multiplied by a fixed binary matrix and then added to a fixed 8-bit vector, shown in Figure 3.5. This step is known as an affine mapping. The resulting mapping is shown in Figure 3.6. Note that no bundle transposition is done on the state during this permutation.

The application of the affine mapping turns an otherwise simple algebraic expression of the S-box into a complex algebraic expression with no fixed or opposite fixed points. This is done to make it harder to use algebraic manipulations to mount attacks on AES. Implementation wise, the implementer is free to implement the S-box as a look-up table, to follow the mathematical

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} \equiv \begin{pmatrix} 10001111 \\ 11000111 \\ 11100011 \\ 11110001 \\ 11111000 \\ 01111100 \\ 00111110 \\ 00011111 \end{pmatrix} \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \quad \text{mod } 2$$

Figure 3.5: Second step in constructing the S-box. Note that $B'_i(x) = A_i^{-1}(x)$. From [10, p. 103].

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

Figure 3.6: AES S-box: Substitution values for the byte xy (in hexadecimal format). From [9].

description, or to implement it in hardware instead of software. The 16 applications of the S-box may also be done sequentially or in parallel. This gives flexibility to AES, as it is easily adaptable to the various needs that arise in the real world.

When decrypting, one uses Sb^{-1}, the inverse S-box instead. The permutation Sb^{-1} is obtained through a two-step operation. First the inverse of the affine mapping is applied. Thereafter, the bytes are mapped to their inverse, same way as when encrypting. This inverse S-box is also called *InvSubBytes*.

3.4.2 ShiftRows

ShiftRows is a bundle transposition. It cyclically rotates row s of the state matrix N_s positions to the left. The values for N_s are simply given as $N_s = s$, so the top row (indexed by 0) is left in place. The row second to top is rotated one position to the left, next one two positions and the bottom

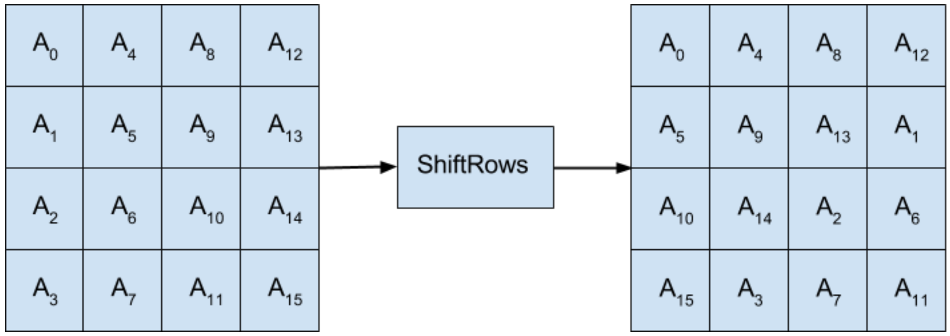


Figure 3.7: ShiftRows rotation of the rows of the state matrix.

row three positions. This ensures that all the new columns contain exactly one byte from each of the previous columns, setting up the stage for MixColumn. See Figure 3.7.

The inverse procedure of ShiftRows, the *InvShiftRows*, rotates the rows 0, 1, 2 and 3 positions to the right, again starting from the top. This simply sets the bytes back to their original positions.

3.4.3 MixColumn

As with SubBytes, *MixColumn* is a bricklayer function. But where SubBytes works on 16 bytes at a time, MixColumn works on a bundle partition of four. Each column in the state matrix is considered a bundle and forms the input to MixColumn. Similar to SubByte, one can process one columns at a time or all four in parallel, depending on the needs of the implementer. As one can see in Figure 3.8, the D-box of MixColumn performs a matrix multiplication between the input column and a fixed matrix. Each element is one of the state bytes, and considered to be an element of $GF(2^8)$.

The elements of the fixed matrix are 01, 02 and 03 and were chosen for their simplicity to implement in software and into dedicated hardware. Multiplying with 01 just gives the element itself, and as mentioned earlier multiplying with 02 is just a left-shift of the coefficients in the respective element. Multiplying with 03 is a left-shift followed by an XOR of the original element. Alternatively, a look-up table for multiplication with 02 and 03 may be used.

Since one input byte influences four output bytes, MixColumn is a major contributor towards the diffusion in AES. Combined with ShiftRows, one byte has influenced all 16 bytes after two rounds of the AES. It also means that the influence of the previous round's AddRoundKey is

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix}$$

Figure 3.8: Overview of the MixColumn D-box. The leftmost vector is the output vector. From [10, p.105].

diffused over four bytes this round, contributing to confusion.

For decryption, the *InvMixColumn* simply uses the inverse matrix given for MixColumn to undo the effect of MixColumn. It should be noted that the first round of decryption undoes the last round of encryption, and therefore no InvMixColumn should be applied.

3.4.4 AddRoundKey

This step is the most straightforward one in AES. The state is modified by XORing it with a roundkey. Since XORing again with the same roundkey is the inverse operation, the only care needed to be taken when decrypting is to ensure that one remembers to add the roundkeys in reverse order.

3.4.5 Key Schedule

The user-selected key in AES is 128, 192, or 256 bits long. The key is partitioned into bytes and arranged in a state with four rows, similar to the cipher block state. The key schedule of the AES treats each column of the state as one bundle. The state itself will have four, six or eight columns, depending on which key size is in use. The next round is then created one column at at time, where the basic concept is that column C_j^r (column j in the r'th round key) is created by XORing C_{j-1}^r and C_j^{r-1}. The exception to this rule is the first column in each round C_0^r. This column is created by XORing C_0^{r-1} with $g(C_b^{r-1})$ where C_b^{r-1} is the last column from round $r-1$.

The function $g()$ is a non-linear function with a four byte input and output, regarded as a column. The four bytes are rotated cyclically one position downwards, and then each byte goes through the S-box. Finally, a round constant is added to the topmost byte of the output column. This round constant is an element of $GF(2^8)$, and Table 3.1 shows the round constants for each

Round	Round Constant
1	00000001
2	00000010
3	00000100
4	00001000
5	00010000
6	00100000
7	01000000
8	10000000
9	00011011
10	00110110

Table 3.1: Round constants for $GF(2^8)$.

round.

The values of the initial state is the given key. Since each round key is 128 bits large, the given key may constitute one subkey, one and a half or two subkeys, for the 128-bit, 192-bit and 256-bit key sizes respectively. This means that the rounds of the key schedule will not necessarily correspond to those of the AES rounds for the 192 and 256 key sizes.

The key schedule for 256 bits is slightly more complicated than for 128 and 192 bits. It introduces another non-linear function $h()$. The $h()$ function takes four input bytes and gives four output bytes by applying the S-box to each of the four bytes. The function $h()$ is applied after the fourth column created in the each key schedule round. Figure (3.9) gives a graphical representation of the 256-bit key schedule. Note however that the indexing direction here is right to left, opposite of the indexing direction used otherwise in this thesis.

3.4.6 Decryption

Decryption in AES is essentially a reversal of the encryption process, using the inverse functions of each step. That means that the order of a normal round starts with AddRoundKey, then InvMixColumn, InvShiftRows and ends with InvSubBytes. Note that the very first round of decryption omits InvMixColumn, and that the very last thing that happens is the application of AddRoundKey, corresponding to the AddRoundKey of the "pre-round" when encrypting.

3.4.7 Design Criterias

AES is designed to be secure, simple, efficient and versatile. Security and simplicity should walk hand in hand, as does efficiency and versatility. As we have seen, many of the choices that makes AES efficient also makes it versatile. The algorithm needs to be as simple as possible, as that makes it easier to analyse and as such indirectly improves the security.

The security aspect is the most important one for any cipher. A good cipher needs to be non-linear, as linear systems are easily breakable. Furthermore, it needs to show resilience towards known cryptanalysis techniques known at the time, especially towards differential and linear cryptanalysis. A cipher also needs to attempt to take future development in cryptanalysis into account. In [13] the two authors of AES explain their reasoning behind their design of AES in more detail, explaining a strategy they call “the wide trail strategy”. It is a recommended read for anyone who wants to know more about AES’ design and thought process.

3.5 Small Scale Variants of the AES

One approach to attacking iterated block ciphers is to define a series of *round-reduced versions* of the cipher, and to see how many rounds one can break. This approach may yield information about potential weaknesses in the cipher at question, as well as information with regards to how large the security margin of that cipher is. Any weaknesses discovered by a round-reduced attack may not apply in the full cipher, but this will give valuable information with regards to where one should look for good attacks. For this thesis, we have chosen to use the fully parameterized framework from [3]. This paper allows for more variation than only reducing the rounds. Also, the small scale versions of the round operations of AES follow the full AES’ design pattern.

3.5.1 Parameters

Two sets of small scale variants, $SR(n,r,c,e)$ and $SR^*(n,r,c,e)$ are defined in [3]. The only difference between them lies in how the final round is defined. In $SR(n,r,c,e)$ the final round is no different than any other round, meaning that the MixColumn permutation is performed, while $SR^*(n,r,c,e)$ follows the AES standard of omitting MixColumn in the last round. The four parameters n, r, c, e are as follows:

- n is how many rounds are performed when encrypting/decrypting. In this thesis we consider $3 \leq n \leq 10$.
- r is the number of rows in the state matrix: $r = 1, 2$, or 4.
- c is the number of columns in the state matrix: $c = 1, 2$, or 4.
- e is the number of bits in a finite field element: $e = 4$ or 8.

Note also that the size of the cipher state no longer is fixed to 128 bits but is now defined as $r \times c \times e$. Indexing of the state array follows the same pattern as with full AES, one column at a time, starting from the topmost element. Normal 128-bits AES is defined as $SR^*(10, 4, 4, 8)$. The cipher blocks produced by $SR(n, r, c, e)$ and $SR^*(n, r, c, e)$ are equal up to ShiftRow in the last round, and the ciphertexts differ only by an affine mapping. Hence, if we can attack one of them we can immediately use the same attack on the other, just by adjusting the key schedule in the last round to compensate for this difference.

Further note that this parameterization does not exceed 10 rounds, and thus only considers keys of 128 bits. Even though it is possible to build upon [3] to include the key sizes of 192 and 256 bits, 128-bit key size is considered a good starting point for our attack.

3.5.2 $GF(2^4)$ and Small-Scale Round Operations

With the introduction of elements in $GF(2^4)$ in addition to the standard $GF(2^8)$ we also need to adopt the round operations to $GF(2^4)$. The polynomial used to define $GF(2^4)$ is $x^4 + x + 1$. For the creation of the S-boxes the same approach as for $GF(2^8)$ is taken. First the inverse of the input is computed, followed by a $GF(2)$-affine mapping. Fig (3.10) shows the S-box summary for both $GF(2^4)$ and $GF(2^8)$. Fig (3.11) gives the resulting look-up table for $GF(2^4)$. There will be as many S-boxes in SubBytes as there are elements in the state array, $r \times c$.

ShiftRows differ from the original only in the fact that there may be less rows and/or less columns to rotate. The matrices to be used in MixColumn depend on the number of rows and the finite field in use. Use Figure (3.12) to choose the right one. Here ρ is a root of $x^4 + x + 1$ and θ is a root of the actual AES polynomial defining $GF(2^8)$.

As with normal AES, we need a total of $n+1$ round keys for all reduced versions, and all small scale variants of AES also begins with a round 0, or "pre-round", where only AddRoundKey is performed. Round keys are added as normal, simultaneously XORing the elements of the subkey array and the corresponding elements of the state array (as elements of $GF(2^e)$).

For the key schedule, a user supplied key of size $(r \times c) \times e$ forms the initial subkey. Each preceding subkey is then derived from the previous one in the same way as for full AES, taking the number of columns c into account. See Figure (3.13). If the finite field is $GF(2^4)$, we use the round constants given in Table 3.2.

Round	Round Constant
1	0001
2	0010
3	0100
4	1000
5	0011
6	0110
7	1100
8	1011
9	0101
10	1010

Table 3.2: Round constants for $GF(2^4)$.

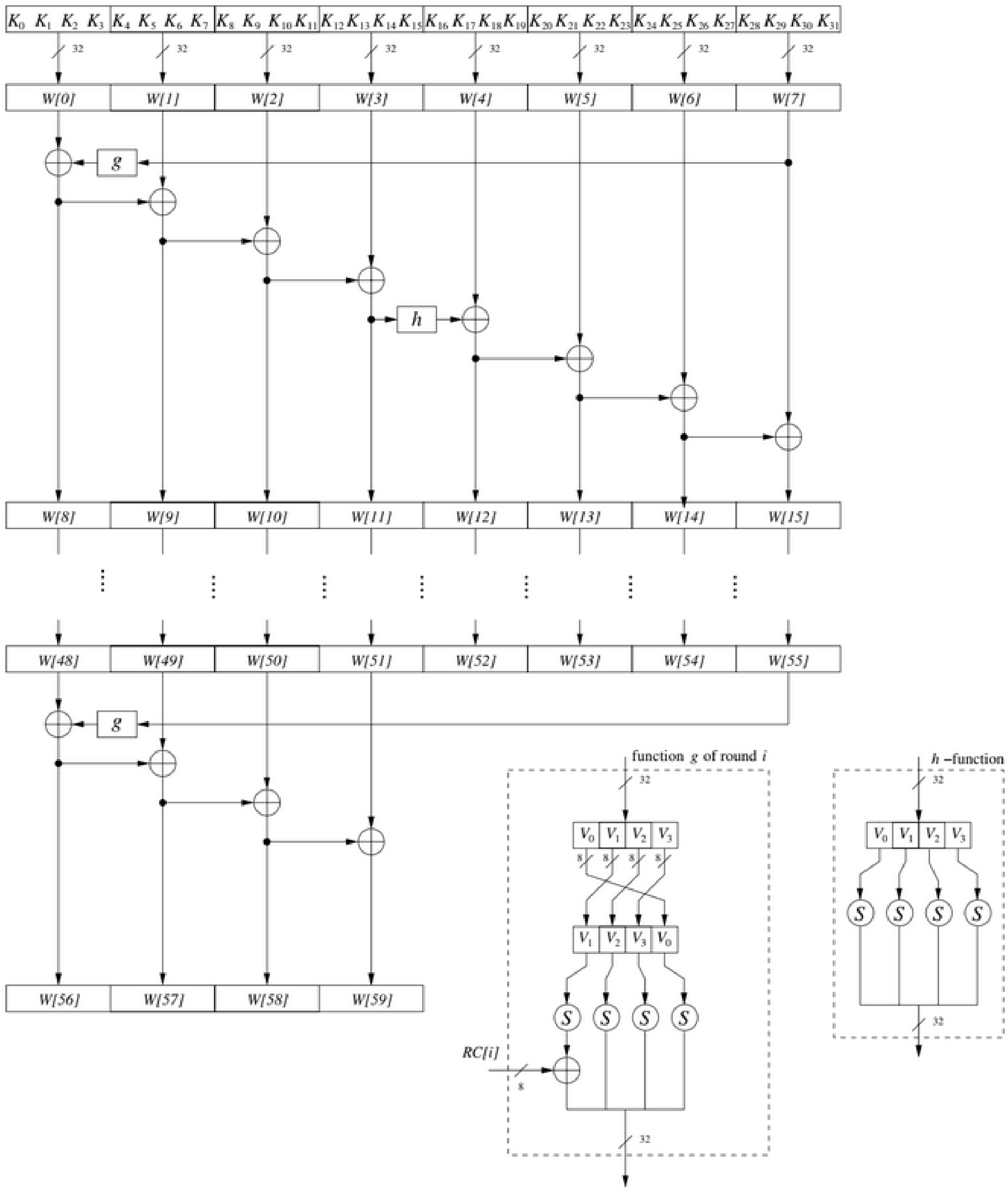


Figure 3.9: AES schedule for 256-bit AES. K_0 to K_{31} are the bytes of the given key. From [10, p.107].

S-Box Summary	$GF(2^4)$	$GF(2^8)$
Irreducible polynomial	X^4+X+1	$X^8+X^4+X^3+X+1$
$GF(2)$-*linear map*	$\begin{pmatrix}1&1&1&0\\0&1&1&1\\1&0&1&1\\1&1&0&1\end{pmatrix}$	$\begin{pmatrix}1&0&0&0&1&1&1&1\\1&1&0&0&0&1&1&1\\1&1&1&0&0&0&1&1\\1&1&1&1&0&0&0&1\\1&1&1&1&1&0&0&0\\0&1&1&1&1&1&0&0\\0&0&1&1&1&1&1&0\\0&0&0&1&1&1&1&1\end{pmatrix}$
Constant	6	63

Figure 3.10: Irreducible polynomial, affine mapping and constant used in creation of the two S-boxes. From [3].

S-Box over $GF(2^4)$																
Input	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Output	6	B	5	4	2	E	7	A	9	D	F	C	3	1	0	8

Figure 3.11: Look-up table for S-box under $GF(2^4)$. From [3].

Number of Rows	$GF(2^4)$	$GF(2^8)$
$r=1$	(1)	(1)
$r=2$	$\begin{pmatrix}\rho+1&\rho\\\rho&\rho+1\end{pmatrix}$	$\begin{pmatrix}\theta+1&\theta\\\theta&\theta+1\end{pmatrix}$
$r=4$	$\begin{pmatrix}\rho&\rho+1&1&1\\1&\rho&\rho+1&1\\1&1&\rho&\rho+1\\\rho+1&1&1&\rho\end{pmatrix}$	$\begin{pmatrix}\theta&\theta+1&1&1\\1&\theta&\theta+1&1\\1&1&\theta&\theta+1\\\theta+1&1&1&\theta\end{pmatrix}$

Figure 3.12: MixColumn matrix for various parameters. From [3].

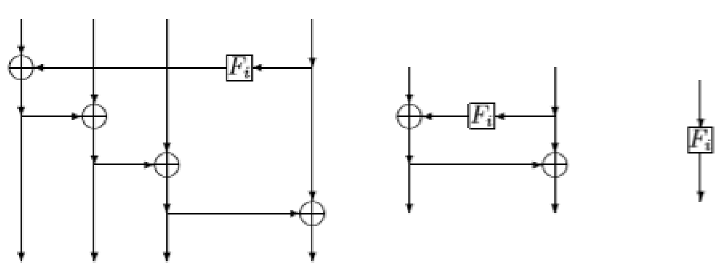


Figure 3.13: One round of key schedule for the different values of c. From [3].

Chapter 4

Multiple Right-Hand Sides and Compressed Right-Hand Sides Equations

Much of the research done in the field of algebraic cryptanalysis has been about SAT-solvers and Gröbner basis computation.

The first task for a researcher in the field of algebraic cryptanalysis is to convert the encryption algorithm in question to a system of polynomial equations. The next part, usually the hardest part, is then to solve these systems of polynomial equations. Many researchers in the field of algebraic cryptanalysis has had, and perhaps still have, high hopes to algorithms based upon Gröbner basis and SAT-solvers. However, even though much research has been done, and much progress made, they have yet to live up to the hopes.

This chapter will use a different algebraic approach. AES, our encryption algorithm in question, will be modeled using linear systems of equations. This is step one, and the hardest part in this step is to model the non-linear S-boxes in a way that works with linear systems of equations. This is solved by introducing the concept of Multiple Right-Hand Sides, which is a technique that opens up for having multiple vectors on the right hand side in a system of linear equations. One needs initially one such Multiple Right-Hand Side for each S-box present. Through mainly a technique called gluing, one is then able to "merge" all this various systems of linear equations into one. Through this process vectors in the right-hand side that would render the equation system inconsistent are identified and removed. This is the topic of the first section.

Unfortunately, the size of such Multiple Right-Hand Sides equation systems tend to grow exponentially during the gluing process, halting the process due to computer memory limitations. In an attempt to remedy this problem, [14] introduces the concept of Compressed Right-Hand Sides (CRHS). This is the topic for the second section. In a CRHS equation, the set of right hand sides is structured as a directed acyclic graph (DAG) instead of a matrix, ordered in levels, and with linear combinations associated with the levels. With some added details, this data structure is known as a Binary Decision Diagram (BDD). CRHS equations can be used to represent encryption algorithms, and with the accompanying techniques for "merging" BDDs and identifying and removing inconsistent paths it is a promising tool for doing algebraic cryptanalysis.

4.1 Multiple Right-Hand Sides

4.1.1 MRHS Equation

An equation on the form

$$\mathrm{A}\mathbf{x} = \mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_s \tag{4.1}$$

is known as a *Multiple Right Hand Side (MRHS) equation* if A is a matrix of size $k \times n$ and rank k, and $b_1, b_2, \ldots, b_s$ are column-vectors of length k. They first appeared in [12]. The vector $\mathbf{x}$ consists of all n unique Boolean variables in use when modelling the cipher, represented as a column-vector with n entries. To simplify notation, we will denote $\mathbf{b_1}, \ldots, \mathbf{b_s}$ as $[L]$ to emphasize that this is no normal system of linear equations. An n-vector $\mathbf{x_0}$ is said to be a solution if and only if it satisfies $A\mathbf{x_0} = \mathbf{b_i}$, for some i and hence a single MRHS equation has at least s solutions, and often much more than that. In this thesis we only consider matrices and vectors over $GF(2)$, though the general principles are applicable for $GF(q)$.

4.1.2 From AES to MRHS Equations

All operations in the AES except for SubBytes are linear so the encryption algorithm has much inherent linearity. The MRHS representation is therefore an efficient way to represent the AES encryption. We will need one MRHS equation for each S-box present in AES (or small-scale variant). This effectively gives us a system of MRHS equation, which we will explain solving strategies for later. To construct the MRHS equations, one must first map variables strategically to bits in the cipher block at various points in the encryption procedure. This must be done in such a way that the bits of the input and output of any S-box can be written as a linear combination of the variables defined.

Figure 4.1 and Figure 4.2 shows what this looks like for the key schedule and the encryption in the SR*(3,2,2,4) small-scale variant. This variant has 16 unkown bits in the user-selected key, denoted $k_0, \dots, k_{15}$. We introduce new variables representing the state at the output of every S-box, except for the layer of S-boxes in the last round. For the three round version these variables are labelled $k_{16}, \dots, k_{39}$, while in the actual encryption we label them $a_0, \dots, a_{15}$ for the S-boxes in the first round and $a_{16}, \dots, a_{31}$ for the second round. The bits in the state output from the S-boxes in the last round can be written as linear combinations of the known ciphertext bits and the last round key, which we already have defined variables for.

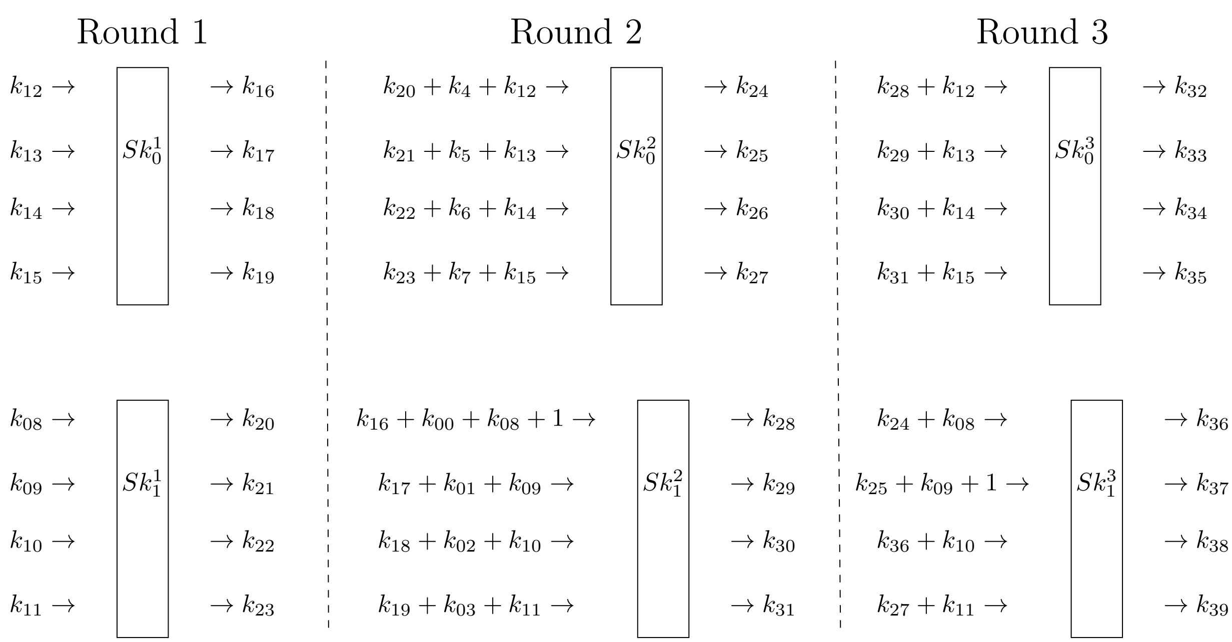


Figure 4.1: S-boxes in the key schedule of SR*(3,2,2,4) with associated variables.

Next we construct one MRHS equation $A_{i,j}\mathbf{x} = [L_{i,j}]$ for each S-box Sk_i^j in the key schedule and each S-box $S_{i,j}$ in the encryption. The input linear combinations of Sk_i^j and $S_{i,j}$ make up the first four rows of each $A_{i,j}$ and the output linear combinations make up the four bottom rows of each $A_{i,j}$. When $e = 4$ each matrix $A_{i,j}$ will be an $8 \times n$ matrix.

Finally, we make a list of all possible inputs to the S-boxes and their corresponding outputs.

Each input/output pair becomes a b-vector in $[L_{i,j}]$. Since every S-box of the AES is the same, every $[L_{i,j}]$ will initially also be the same. The vector $\mathbf{x}$ contains all unique variable present across all $A_{i,j}$'s.

The set of different MRHS equations leaves us with a system of MRHS equations that describes the whole encryption process. For the SR*(3,2,2,4) example in figures 4.1 and 4.2 we get 18 MRHS equations in 72 variables, while the full 128-bit AES, would leave us with 200 MRHS equations; 160 from the encryption process itself and 40 from the key schedule.

4.1.3 Solving a System of MRHS Equations

We are given a system of MRHS equations

$$A_1\mathbf{x} = [L_1], \ldots, A_m\mathbf{x} = [L_m] \tag{4.2}$$

where A_i and $[L_i]$ are matrices with k_i rows. The matrix A_i has n columns, and the number of columns in L_i is s_i. Not all variables appear in all equations, and the related columns in the A_i's are zero. A solution to (4.2) is an assignment to the variables of $\mathbf{x}$ such that $\mathbf{x}$ is a solution to every MRHS equation in (4.2).

A column in an $[L_i]$ that is never produced for any solution to (4.2) may be thought of as *wrong*, while a column in $[L_i]$ that is produced for a solution to (4.2) may be thought of as *right*. The challenge when dealing with systems of MRHS equations is to identify and remove wrong columns while keeping only the right ones. When a solution $\mathbf{x_0}$ is identified, we can simply look-up the values for the key variables k_i in $\mathbf{x_0}$, and we have found a valid key. This is the same as breaking the cipher.

For one given plaintext/ciphertext pair, some ciphers may have more than one key that encrypts the given plaintext into the given ciphertext. The likelihood of this occurring is related to the bit-size of the key and the plaintext. If the block size is larger than the key size we have more constraints than free variables in the system. In this case, with large probability there is only one solution to the system, and the key can be determined uniquely. If the block size is smaller than the key size we do not get enough constraints to uniquely determine the key, and solving (4.2) will give a whole set of possible keys. When the block and key sizes are equal, we may or may not

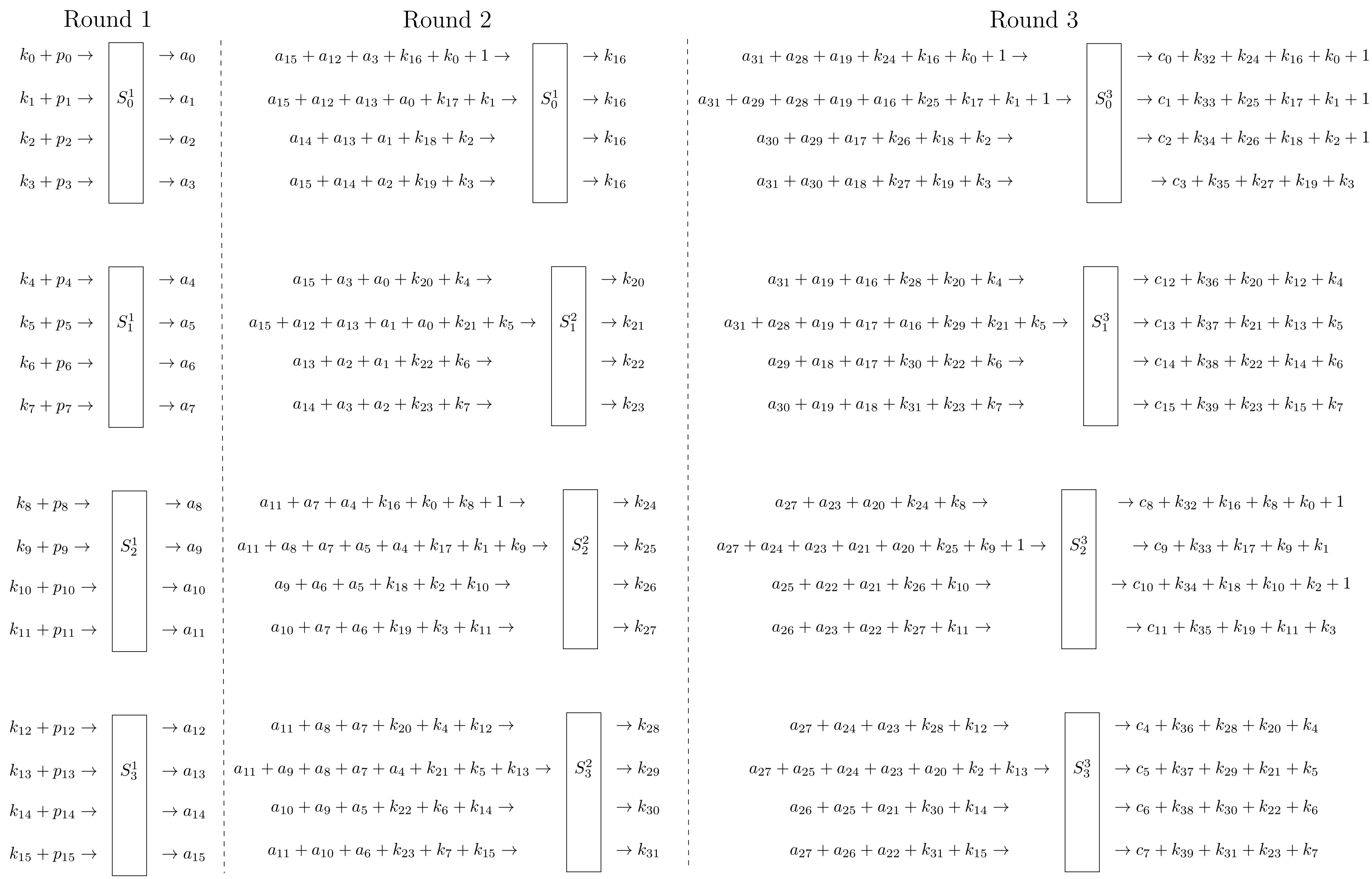


Figure 4.2: SR*(3,2,2,4): S-boxes in the encryption, with associated variables.

get unique solutions. For the small-scale AES variants with equal block and key sizes we have often seen that more than one key fit in a system defined by a given plaintext/ciphertext pair.

Let $A_i\mathbf{x} = [L_i]$ and $A_j\mathbf{x} = [L_j]$ be two MRHS equations in the system of MRHS equations. There are two main operations used when trying to solve a system of MRHS equations.

Agreeing: The first operation is called *agreeing*. For a thorough explanation of agreeing, see[12]. The idea behind agreeing is to temporarily merge two MRHS equations. Then, by identifying any linear dependencies among the rows of the two matrices A_i and A_j, we may identify some wrong columns in $[L_i]$ and $[L_j]$, and remove them. If there are no linear dependencies then no columns in $[L_i]$ or $[L_j]$ can be shown to be wrong, and agreeing will give us no new information towards a solution. Otherwise, we examine all pairs $\mathbf{b_i}, \mathbf{b_j}$ where $\mathbf{b_i} \in [L_i], \mathbf{b_j} \in [L_j]$. If the joined right-hand side $(\mathbf{b_i}, \mathbf{b_j})$ does not give a consistent system for the joined $[A_i, A_j]$-matrix, then $\mathbf{b_i}$ and $\mathbf{b_j}$ do not agree. At least one of them must be a wrong right-hand side. If $\mathbf{b_i}$ does not agree with *any* of the columns in $[L_j]$, then certainly $\mathbf{b_i}$ must be wrong and can be safely deleted, and the same of course applies to $\mathbf{b_j}$ and $[L_i]$. The MRHS equations are then "decoupled", meaning that both A-matrices revert back to their original form but with possibly fewer right-hand sides in the MRHS equation. They are now free to agree with other MRHS equations.

All MRHS equations of AES and small-scale AES starts out in an agreed state, meaning that no two inital MRHS equations agreed together will identify any wrong columns. In order to make any progress towards finding a solution we then need to use *gluing*.

Gluing: We will present the general idea behind gluing here, as this will make some of the observations later more apparent. For a more mathematical view, we refer to [12]. The purpose of gluing is to permanently merge two MRHS equations into one MRHS equation. In this process, any linear dependencies will be solved, removing any wrong columns b_k in the resulting merged $[L]$.

Gluing A_i and A_j into A is straightforward, stack A_i on top of A_j. Then make a list $[L]$ of right hand sides where each column of $[L_i]$ is paired with each column of $[L_j]$. If $[L_i]$ had n_i columns and $[L_j]$ had n_j columns, then $[L]$ contains $n_i n_j$ columns. If there are linear dependencies among the rows of A, we find all columns in $[L]$ that gives an inconsistent system and remove them as they must be wrong. We then get the glued MRHS equation $A\mathbf{x} = [L]$. We create $[L]$ the way we do since any present column b_i in $[L_i]$ could be a solution to $A_i\mathbf{x} = [L_i]$, any present column b_j in $[L_j]$ could be a solution to $A_j\mathbf{x} = [L_j]$, and thus any combination of b_i and b_j could be a solution to $A\mathbf{x} = [L]$. Lastly we check for any linear dependencies and remove inconsistent columns in $[L]$.

4.1.4 Size of MRHS Equations After Gluing

Since AES only uses one S-box, all MRHS equations will have the same initial right-hand sides (RHS). For the first and last round, plaintext and ciphertext constants will then alter their respective RHS's. Each matrix starts off as a $2e \times n$ matrix, where n is the total number of unique variables of the system. Every $[L_i]$ will start off with 2^e columns, and each row in $[L_i]$ has an equal number of 0's and 1's since the S-box is a permutation. XORing in constants does not change this, as each value in each row is XORed with the constant. As XORing rows may be done as a binary function recursively as many times as we want, we see that we may expect half of the columns in $[L]$ to be inconsistent when solving a linear dependency. Let s_i be the cardinality of $[L_i]$ and s_j the cardinality of $[L_j]$. Then gluing these RHS's as part of a gluing operation will result in $2^{s_i} \times 2^{s_j}$ columns before removing columns due to inconsistencies. Let $\Delta(s_i, s_j)$ be the number of linear dependencies in the corresponding glued matrix A. Then we may expect

$$\frac{2^{s_i+s_j}}{2^{\Delta(s_i,s_j)}} = 2^{s_i+s_j-\Delta(s_i,s_j)} \tag{4.3}$$

columns in $[L]$ after solving the linear dependencies. We see that the size of $[L]$ is expected to grow when $s_i + s_j > \Delta(s_i, s_j)$, stay the same when $s_i + s_j = \Delta(s_i, s_j)$ and finally shrink when $s_i + s_j < \Delta(s_i, s_j)$.

We may use this to calculate how many columns we may expect in the final $[L]$ after gluing all MRHS equations together. In the case of the full AES, we have 1600 unique variables and 3200 rows in the final A, conceived after gluing all 200 MRHS equations together. This gives us (at least) 1600 linear dependencies since there are 1600 more rows than columns in the final A. The equation:

$$\frac{2^{8\times 200}}{2^{1600}} = 2^{1600-1600} = 1 \tag{4.4}$$

tells us that we can expect a unique solution when all gluing operations finish.

Even though this seems promising, (4.3) tells us that when any $[L]$ grows due to gluing, it grows exponentially, and needs exponentially more memory to store all the RHS's. In an attempt to circumvent this problem, [14] introduces the concept of Compressed Right-Hand Sides.

4.2 Compressed Right-Hand Sides

A Compressed Right-Hand Side (CRHS) system [14] is a MRHS system where the right-hand sides are stored using a binary decision diagram instead of multiple independent vectors. Both CRHS and MRHS represent the same abstract concept of modeling a cipher. Where MRHS displays more explicitly the changes done to the right-hand sides of the system, CRHS will have less or equal memory requirements for large numbers of right-hand sides. Since a BDD is very different as a structure than a matrix, the gluing operation from MRHS must be replaced: To remove any linear dependencies within a BDD the concept of linear absorption will be introduced.

Where MRHS was developed for crytpanalysis, BDDs were designed for other purposes and see a wide variety of use in the computer science community [5]. In this thesis a BDD is understood as a way to represent the right hand sides of a MRHS system, and as such will differ somewhat from the traditional understanding. Where traditionally only a single variable may be associated with each *level* of a BDD (explained below), we allow for linear combinations to be associated as well.

4.2.1 Binary Decision Diagrams and Compressed Right-Hand Sides Equations

A Binary Decision Diagram (BDD) is a Directed Acyclic Graph (DAG) with exactly one source node and one sink node. The nodes, except the sink, have at least one and at most two outgoing edges, named the 0-edge and the 1-edge. As the name suggests, these represents the values 0 and 1. All nodes except the sink are considered internal nodes. If the source is at the top and the sink at the bottom, all the remaining nodes are in between, and all the edges are directed downwards. The nodes are arranged in levels, with no edges between any two nodes of the same level. Each internal level, that is all levels except the sink level, has either one variable, or a linear combination of variables, associated with it. See Figure 4.3 for an example.

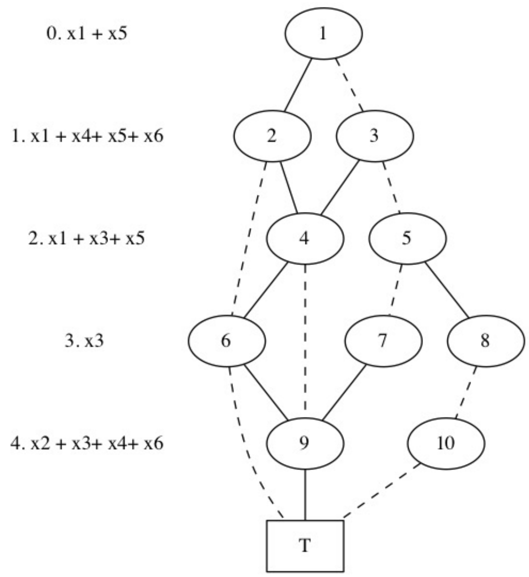


Figure 4.3: BDD with 5 levels.

Choosing an outgoing edge from the associated level is the same as assigning that edge's value to the variable/linear combination. A BDD with $k+1$ levels has k variables or linear equations associated with it, there is nothing assigned to the bottom level with only the sink node in it. A path through the graph from source to sink assigns values to all levels, and is k edges long. This path then may be thought of as equivalent to a column vector in $[L]$ of a MRHS equation, and all paths in the graph is then equivalent to some $[L]$. The associated variables and linear equations may be though of as the matrix A. Then we have:

Definition 7 (CRHS equation. [14])**.** A Compressed Right-Hand Side equation is written as $A\mathbf{x} = D$, where A is a binary $k \times n$-matrix with rows $l_0, \ldots, l_{k-1}$ and D a BDD with (from top to bottom) $l_0, \ldots, l_{k-1}$ attached to the levels. Any assignment to $\mathbf{x}$ such that $A\mathbf{x}$ is a vector corresponding to a path in D, is a satisfying assignment. If C is a CRHS equation then the number of nodes in the BDD of C is denoted $B(D)$.

Any assignment to $\mathbf{x}$ such that $A\mathbf{x}$ gives a path in D may be though of as *right*, while assignments $A\mathbf{x}$ giving a vector that is not a path in D may be thought of as *wrong*.

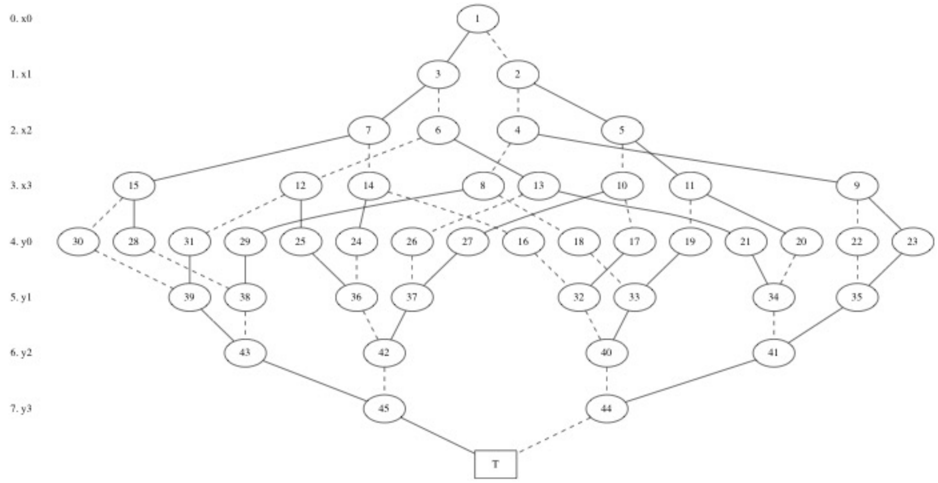


Figure 4.4: The BDD representing a 4-bit S-box.

4.2.2 BDD Construction

The construction of a CRHS based upon an S-box with n input bits and m output bits follows the same pattern as for MRHS. The BDD itself is based upon the substitutions defined in the S-box and will have $n+m$ levels. This is constructed as follows: Create a complete binary tree from the top node based on the 2^n possible input values of the S-box. The source node is associated with the LSB of the input. Then build a "reverse", or bottom-up, complete binary tree based on the 2^m possible output values, from the sink node. The final step is then to link the two binary trees, which is done based on the substitutions defined by the S-box. Each internal level has their appropriate variable or linear combination associated with it. Figure 4.4 shows the BDD of a 4-bit S-box.

4.2.3 Solving Systems of BDDs: Merging BDDs

As with MRHS, one CRHS equation is created for each S-box in the cipher, and as with MRHS we get a system of CRHS equations describing the whole cipher:

$$A_0\mathbf{x} = D_0, A_1\mathbf{x} = D_1, \ldots, A_m\mathbf{x} = D_m \tag{4.5}$$

A solution to 4.5 is an assignment to the variables of $\mathbf{x}$ such that every CRHS equation in 4.5 has a solution. There is no concept of agreeing in CRHS, but CRHS equations may be merged in a similar fashion to gluing. Merging two CRHS equations into one is even simpler than for MRHS: If T_0 is the terminal node for D_0 and U_1 is the source node for D_1, remove T_0 from D_0 and let all the edges that used to go to T_0 instead go to U_1. Now the two BDDs are connected and represent the BDD D for a single CRHS equation. This joining operation does not produce any new nodes (in fact, it removes one), and we see that $B(D) = B(D_0) + B(D_1) - 1$. This new CRHS equation has all combinations of paths from D_0 and D_1, without needing any additional memory!

As it is possible to hold all 200 BDDs from full AES in memory at once, we may easily build a single CRHS equation that represents the full AES. Unfortunately, the BDD of this CRHS equation would hold 2^{1600} paths, where all but maybe one or two are wrong. We need a way to remove the wrong paths.

4.2.4 Tools for Solving CRHS Equation System: Swapping and Adding Levels

Where matrix operations for resolving linear dependencies and identifying wrong vectors is fairly speedy in MRHS, identifying and removing wrong paths in a BDD is more demanding. The solution to this issue is known as linear absorption, and was introduced in [15]. Linear absorption is comprised of two subroutines, *swapping* and *adding* levels. It also uses the fact that for a fixed order of linear combinations, there exists a unique reduced BDD. Running the *reduction algorithm* [2] on a BDD removes unnecessary nodes and reduces a BDD to its unique state. Simplified, the reduction algorithm will absorb nodes and paths representing equivalent paths, essentially removing duplicates. It will also look for internal nodes (except the source) that has no incoming or outgoing edges. These will be removed, as they are not parts of complete paths. For the rest of this work we will assume that a BDD may always be reduced, and that the reduction algorithm is run whenever necessary.

In order for any linear dependencies to be resolved in a CRHS equation, the linearly dependent rows and corresponding levels in the BDD needs to be adjacent. In order to achieve that, we need to either “bubble” lower levels, or “sink” higher levels into position. This is achieved through the *swapping* subroutine. When swapping two adjacent levels, we must ensure that we do not lose nor add any new paths. Swapping achieves this by clever rebinding of the involved nodes and edges, and when done, the two involved levels have swapped places, without changing the solution space of the CRHS equation. This is then repeated until the level we wish to move has arrived where we want it.

Where swapping rearranges the BDD and corresponding matrix, the subroutine for adding two levels together replicates the XORing of two rows in a MRHS equation. As with swapping, we must keep the solution space intact in the process. The addition of levels needs the two levels to be adjacent. Then it will XOR a copy of the highest level onto the level below through clever rebinding of the involved nodes and edges. This process resembles that of swapping, although the rebinding follow different rules.

The drawback of the operations of swapping and adding levels is that new nodes usually needs to be created during the process. Hence the memory requirements grow when applying these operations. The memory increase is not as dramatic as with gluing MRHS equations though, and we can solve larger systems in practice using the CRHS representation than we can with MRHS representation.

4.2.5 Resolving Linear Dependencies in BDDs: Linear Absorption

We are now ready to utilize linear absorption to get rid of the wrong paths. Let $(l_0, l_1, \dots, l_{k-1})$ be the ordered set of linear combinations associated with the levels. Assume that $l_{i_1} + l_{i_2} + \cdots + l_{i_r} = 0$ is a linear dependency, where $i_1 < i_2 < \cdots < i_r$. We can then utilize swap repeatedly, moving l_{i_1} to just above l_{i_2} before using level addition to replace l_{i_2} with $l_{i_1} + l_{i_2}$. Then we utilize swap again to move $l_{i_1} + l_{i_2}$ to the level just above l_{i_3} and replace l_{i_3} with $l_{i_1} + l_{i_2} + l_{i_3}$. We then keep repeating this process, picking up each l_{i_j} along the way, until we have replaced l_{i_r} with $l_{i_1} + l_{i_2} + \cdots + l_{i_r}$. We call this level a zero-level, because the linear dependency indicates the linear combination for this level is $\mathbf{0}$. Then we may remove any 1-edges out of the zero-level, as any path that choose any 1-edge would make the CRHS equation inconsistent via the $\mathbf{0} = 1$ assignment. This leaves us with a level with only outgoing 0-edges. There is no longer any choice to be made for any path going through the zero-level, and thus we may redirect any incoming edges to the correct nodes in the level below. We can then delete all nodes on the zero-level and the corresponding $\mathbf{0}$-row in the matrix, decreasing the number of levels in the CRHS by one. We say that the linear dependency $l_{i_1} + l_{i_2} + \cdots + l_{i_r} = 0$ has been *absorbed*.

We can repeat this process, absorbing one linear dependency at a time, until all linear dependencies in the CRHS equation has been absorbed. Any remaining path in the BDD will now yield right-hand sides that give a consistent linear system, which can readily be solved.

4.2.6 Complexity

Experimental results on the cipher Trivium suggests that the number of nodes in the BDDs grow very slowly when absorbing the first linear dependencies, but increase more rapidly when fairly large BDDs are joined, with many linear dependencies in them [15]. Though from (4.4) we know that we may expect one path in the BDD when all CRHS equations have been merged and all linear dependencies absorbed. A one-path BDD only has $n+1$ nodes, which is very small. This tells us that there exists some tipping point where the number of nodes is at a maximum, and must start to decrease when further absoptions and reductions are applied. At this tipping point the BDD will contain its maximum number of nodes, and we will use this maximum, or peak, number as our measure of memory complexity in the next section.

4.3 Order of Joining

When and how many linear dependencies arise in joined CRHS equations relies heavily on the order the CRHS equations are joined in. The optimal order for joining CRHS equations (and MRHS equations) is an unsolved problem. However, in [11] three strategies are proposed:

Automatic Ordering. This strategy can be applied to any system, and is as such reckoned as a default strategy. It does not require any knowledge of how the CRHS equations has been made. This procedure looks for the subset of CRHS equations that contains the smallest number of nodes, while still having linear dependencies. The CRHS equations of this subset is subsequently joined and the linear dependencies absorbed. This is a greedy approach that always tries to make a minimal CRHS equation to absorb dependencies in, which should then not become too big after absorption. This will decrease the number of CRHS equations by at least one, and we continue this process until we only have one CRHS equation left with no linear dependencies.

Divide-and-Conquer. The thought behind this strategy is that it is always easier to join two big CRHS equations and absorb their dependencies when there are *few* dependencies. The assumption is that we will only have a minimum of dependencies left when the last (maybe big) BDDs are forced to be joined together. Basically, the earlier we can absorb linear dependencies the better, as this would keep the number of nodes low since we absorb most dependencies only in small BDDs.

This strategy proposes to divide the system into two roughly equally sized halves, where we want as few linear dependencies between CRHS equations in opposite halves as possible. We may do this recursively on the halves, ending the splitting ideally when the "half" only contains one or two original CRHS systems. These halves are then joined and all dependencies absorbed. This will leave us with only one CRHS equation with no dependencies in each half. We then join the next halves, and absorb the relatively few remaining linear dependencies. This is repeated until all CRHS equations are joined and all dependencies absorbed.

The challenge is to find the optimal way to split a system into two equally sized parts. This seems to be a hard problem, and knowledge of the cipher should be utilized.

Finding Good Joining Order by Cryptanalysis. This strategy is to use knowledge of the cipher represented by the CRHS system to decide a good order to join the equations. The order of the joining should be such that each linear dependency only involves linear combinations on levels that are relatively close to each other. This makes the absorption process somewhat local, which should help keeping the complexity down.

Chapter 5

Experiments and Findings

This thesis considers a particular branch of cryptanalysis known as *algebraic cryptanalysis*, applied to small-scale variants of the AES. We build on earlier work done in [3, 12, 11], where a few small-scale AES systems have been tried solved using different techniques. In [3] MAGMA with its implementation of the F4-algorithm was used, while in [12] and [11] Multiple Right-Hand Sides and Compressed Right-Hand Sides were used. Here we extend the results from these papers by trying to solve many systems from almost all SR-variants with the latest version of the software made for solving equation systems in the CRHS representation.

These experiments seek to gain knowledge about what the complexity is for solving these types of systems, and how it varies in the different systems. In particular, we try to answer the following questions:

- How does the memory complexity vary with different rounds in a small-scale variant of AES?
- How does execution time vary with rounds in a small-scale variant of AES?
- Are there differences in solving complexity in variants with the same key size, but with different state array dimensions?

5.1 Setup

5.1.1 Software

For each version of small-scale AES with bundle size of 4 bits, a set of eight fixed values were used for both plaintext and key. The respective values for 16-, 32- and 64-bit systems are given in Appendix A. These strings were not carefully chosen. We just wanted some control over the plaintext and keys hoping that something interesting would come out of it.

We limited ourselves to only consider variants with 4-bit S-boxes. This still gives many SR-variants to consider, most of which give systems that are hard to solve on the computer resources available for these experiments.

For 16-bit keys, we considered block arrays of sizes 1×4, 2×2 and 4×1. For 32-bit keys there are two variants of cipher blocks, namely 4×2 and 2×4. For 64-bit keys there is only the 4×4 state to consider. For each of these we made systems for 3 to 10 rounds, i.e. 8 variants for each state array. With both plaintext and key taking all different combinations from a set of 8 values means that we can generate 64 instances for each small-scale variant. With both SR and SR* this gives a grand total of $(3+2+1)*8*64*2 = 6144$ instances.

We had limited access to hardware to run the attack on, and it was therefore decided to only run the SR* versions. The SR* versions were chosen as full AES itself is a SR* version. This left us with 3072 instances to run. All the different system instances were generated by Java code developed by myself.

The actual attacks on the instances were carried out by C-code developed by Raddum over the course of 10 years. This code has been used in previous work [14]. The solving strategy of this C-code uses the “automatic ordering” gluing strategy, which join BDDs that has the fewest number of nodes, while still containing linear dependencies. Furthermore, a limit to the maximum node count of any BDD was set to 2^{26}. With the size of each node in this C-code, that means a memory limit of roughly 8 GB. No two BDDs would be glued together if their combined node count exceeded 2^{26}. Also, if the node count in a BDD exceeds 2^{26} after absorbing a linear dependency, the program would abort and report "not solved". If no BDDs can be glued together without exceeding this limit, the attack would be aborted and considered “not solved”. If the memory limit is not reached, the program will always solve the system within reasonable time.

5.1.2 Hardware

For the execution of the attacks a virtual machine (VM) was borrowed from UH IaaS (http://www.uh-iaas.no/), an organization that provides infrastructure-as-a-service for academia in Norway. The exact specifics of the underlying hardware are unknown, but we know that we were given a VM with 8 GB of RAM.

5.1.3 Attack Execution

The whole process was controlled by a master script, written in Python for this purpose. After setting up the output files, one .txt and one .csv, it would call the C-code with a system instance as parameter. This would instantiate a so-called *run-through*. A run-through would run in its own thread and execute an attack on the given instance. Upon completion of the attack, but before returning control to the master script, the instance identifiers, time consumption, solving complexity in terms of nodes, as well as 1 for "solved" or 0 for "not solved", would be written to the output files. This process was then repeated for all instances, and progress could be monitored through a log file maintained by the master script.

5.1.4 Known Sources of Error

First, a misconfiguration in the master script meant that no $n = 10$ variants were run, so we only ran 2688 instances instead of the intended 3072.

For reasons that are not completely known, many run-through results have not been written to the output files, despite being logged as instantiated by the master script. We can not be certain of the explanation of this behavior, but it is our belief that they were killed by the VM when a run-through required more than 8 GB of RAM. We do have some reported results which required slightly more than 8 GB of memory, but our belief is that when exceeding the 8 GB of RAM, the constant read and write to the Hard Disk Drive (HDD) could violate some other restriction and result in killing the process.

Our belief is primarily based upon two things: This behavior has not been experienced before in this C-code's history. Furthermore, in January we were able to repeat the experiment for one of the most affected system variants on a laptop with 16 GB memory and an Intel i7 processor, without any loss. (These results are not included in this thesis as they are not directly comparable to the initial results). This indicates that the issue lies with the VM and not with the C-code.

We have not compensated for the loss of instances in the results and discussion, unless otherwise stated.

5.1.5 Potential Sources of Error

A natural potential source of errors would be the software itself, both through design flaws and bugs, although we have taken steps to mitigate this risk. The C-code has been developed and tested for more than ten years, and some of its results have been verified by others. The generation of instances in Java have both utilized JUnit testing, as well as comparing output of certain variations with already known instances used in earlier work. The master script was never involved in the actual attacking, it only administered the order of run-throughs.

A second potential source of error would be the virtual RAM of the VM, if such exist. Virtual RAM is a feature in which less frequent used data in RAM is written to the HDD when approaching the limit of the actual RAM. The computer would swap data back and forth between RAM and HDD as needed, usually without the user noticing. However, reading and writing to HDD is significantly slower than reading and writing to RAM. That means that if a BDD or system of BDDs grew above 8 GBs, constant reading and writing to HDD would be needed, significantly slowing down the attack and reporting inflated numbers for execution time. As we do not know the internal handling procedure of the VM, nor even if virtual memory was enabled, it is hard to asses the significance of this issue.

Last, we have no way of knowing how, or even if, varying workload of the underlying software and hardware would affect the run-time of the various run-throughs.

5.2 Findings and Results

5.2.1 Reported and not Reported Instances

As pointed out in 5.1.4, many instances have not reported their metrics. We think of them as "lost". Out of 2688 instances, we did not get reports on 1313 of them. This gives us a 51 percent report rate. These will be treated as not solved, however they will be kept out of all metrics, unless otherwise stated.

64-bit systems have the largest number of lost instances, with only two instances reported of 512 expected, giving ≈ 0.4 percent reporting rate. For the 32-bit systems 842 out of 896 expected are reported, which gives ≈ 94 percent reporting rate. The 16-bit systems come in between, with 531 reported instances out of the 1344 expected, for an approximate 40 percent reporting rate.

5.2.2 64-bit Systems

We have two reported instances, and not surprisingly, none of them were solved. They are both of the SR*(3,4,4,4) variant. Their average run-through time was 110 seconds and both exceeded the 2^{26} node limit. Both systems were generated using the same plaintext value. See Figure 5.1.

Sys. Variant	ID	Time (s)	Average of Nodes (log2	1 if solved	plaintext	ciphertext	key
SRstar(3,4,4,4)	-837287465.bdd	118	26,034	0	ffffffff00000000	7c17471ab3fd9edc	174ca832174ca832
SRstar(3,4,4,4)	2017809514.bdd	102	26,022	0	ffffffff00000000	9e176f39d47fe4bb	00000000ffffffff

Figure 5.1: Data for the 64-bit system instances.

5.2.3 32-bit Systems

Here we have 842 reported instances, out of 896 expected. Six variants lost no instances, and among the others only one system has lost more than 10 instances. Out of the 842 reported instances, 9 were solved. All the solved instances were of the SR*(3,2,4,4) system variant. The average times of the run-throughs vary between 56 and 402 seconds. Figure 5.2 shows the average time and number of nodes for each individual system variant, while Figure 5.7 shows the distribution of solved/non-solved instances based on the plaintext used to generate the instances.

Sys. Variant	Reported Instanc	Nr. of solved	Average of Time (s)	Average of Nodes (log2)
SRstar(3,2,4,4)	64	9	93,41	24,188
SRstar(3,4,2,4)	64	0	55,80	24,748
SRstar(4,2,4,4)	64	0	177,98	24,642
SRstar(4,4,2,4)	61	0	143,26	24,466
SRstar(5,2,4,4)	64	0	194,39	24,648
SRstar(5,4,2,4)	62	0	133,55	24,459
SRstar(6,2,4,4)	64	0	168,83	24,528
SRstar(6,4,2,4)	62	0	244,10	24,680
SRstar(7,2,4,4)	58	0	307,69	24,814
SRstar(7,4,2,4)	63	0	322,84	24,676
SRstar(8,2,4,4)	54	0	353,20	24,483
SRstar(8,4,2,4)	64	0	374,36	24,684
SRstar(9,2,4,4)	42	0	243,45	24,902
SRstar(9,4,2,4)	56	0	478,02	24,360
Grand Total	**842**	**9**	**231,06**	**24,585**

Figure 5.2: Metrics for the 32-bit system variants.

5.2.4 16-Bit Systems

We have 531 reported instances out of 1344 expected. We have 15 system variants that contains reports, which leaves 6 variants that lost all instances. We solved in total 482 instances, which is 90.7 percent out of the reported instances and 35.9 percent of the total instances. Figure 5.3 gives details of the metrics for the 16-bit system variants.

5.3 Discussion on Findings

The first thing we notice, is that whether the 64 systems within one variant are solved or not actually depend on the fixed values of the plaintext and the key. This may be a feature of the particular solving strategy used, where some of the constants from the plaintext and ciphertext can give more "structure" or "order" in some of the initial BDDs, and hence fewer nodes in these. So different plaintexts/ciphertexts may lead to different orders for joining BDDs, as always the BDDs with fewer nodes are preferred, and some joinings may turn out to be more fortunate than others.

Other observations are in large part based on the instances with a 16-bit block, as these are mostly the ones where we were able to solve anything.

Sys. Variant	Reported Instances	Nr. of Solved	Average of Time (s)	Average of Nodes (log2
SRstar(3,1,4,4)	64	64	4,02	17,176
SRstar(4,1,4,4)	64	64	65,34	21,910
SRstar(5,1,4,4)	64	64	167,88	22,779
SRstar(6,1,4,4)	48	48	559,46	23,867
SRstar(7,1,4,4)	10	2	637,70	26,001
SRstar(8,1,4,4)	2	0	450,50	26,001
SRstar(9,1,4,4)	1	0	785,00	26,060
SRstar(3,2,2,4)	64	64	48,58	21,322
SRstar(4,2,2,4)	64	64	388,56	23,456
SRstar(5,2,2,4)	56	56	867,07	23,882
SRstar(6,2,2,4)	52	50	965,75	24,112
SRstar(7,2,2,4)	32	2	1068,06	26,080
SRstar(8,2,2,4)	4	4	3395,25	24,792
SRstar(9,2,2,4)	0	0	-	-
SRstar(3,4,1,4)	1	0	330,00	26,060
SRstar(4,4,1,4)	5	0	468,60	26,031
SRstar(5,4,1,4)	0	0	-	-
SRstar(6,4,1,4)	0	0	-	-
SRstar(7,4,1,4)	0	0	-	-
Srstar(8,4,1,4)	0	0	-	-
SRstar(9,4,1,4)	0	0	-	-
Grand Total	**531**	**482**	**428,03**	**22,580**

Figure 5.3: Metrics for the 16-bit system variants.

5.3.1 SR*$(n, 4, 1, 4)$

The SR*$(n, 4, 1, 4)$ variants stand out, as four of the missing six variants are all from this set, for $n > 4$. For $n = 3$ and $n = 4$ we have 1 and 5 reported instances, respectively, none of which were solved. The fact that the instances with 4×1 block were the hardest to solve is not so surprising when one looks into how the encryption algorithm works in this case.

When the cipher block only has one column we get full diffusion after only one round of encryption, instead of the two rounds needed for normal AES. Moreover, the key schedule becomes fully non-linear as every column in the round keys pass through the $g()$-function when producing the next column. This gives more variables in the key schedule than the other instances of the same bit size.

5.3.2 SR*(n,2,2,4) and SR*(n,1,4,4)

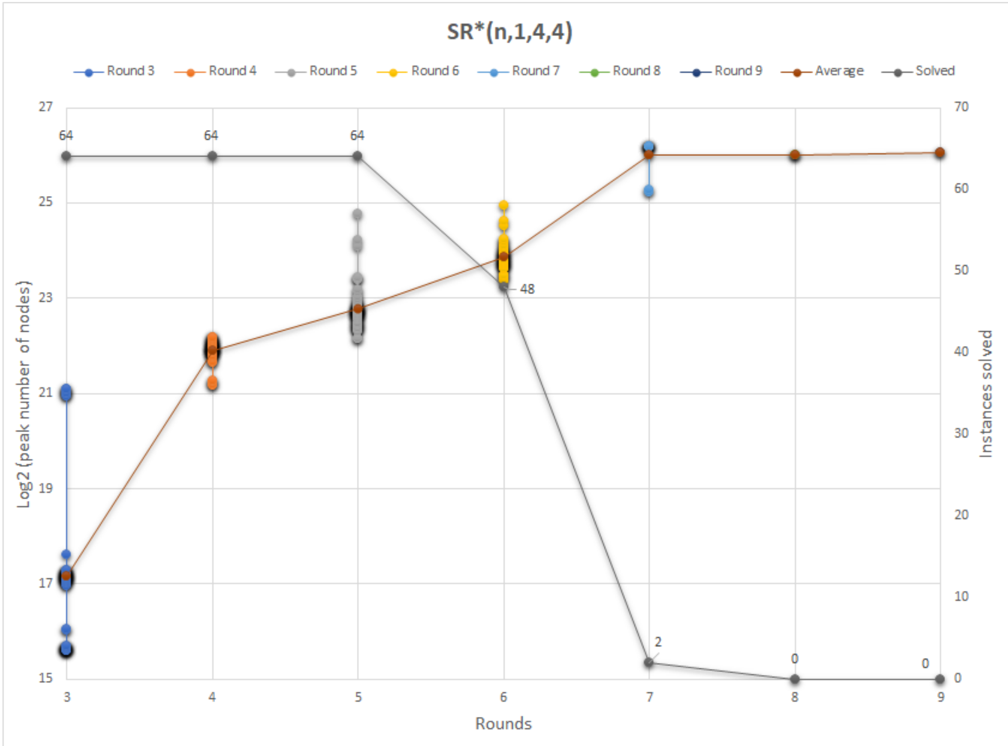
For the other two 16-bit block variants, SR*(n,2,2,4) and SR*(n,1,4,4), we see a change for $n \geq 7$. The fraction of reported instances as well as solved instances drops, from an average 59.5 percent reported and 59.25 percent solved to an average 9.8 percent reported and 1.6 percent solved. Also, up until round seven, all reported instances but two are solved. So with the 2^{26}-limit on complexity we can solve most 2×2 and 1×4 instances up to seven rounds, but for higher number of rounds the solving complexity is very often too high for our limit.

There appears, however, not to be a big difference between the hardness of solving SR*(n,2,2,4)-instances and SR*(n,1,4,4)-instances, in the sense that we can solve roughly the same fraction of instances, see Figure 5.4. This is maybe more surprising as the variants with 1×4 block are rather degenerate compared to those with 2×2 block. The encryption algorithm with a 1×4 block does not give any diffusion at all as both ShiftRows and MixColumn become the identity mapping. Hence the variants with a single row in the cipher block state are vulnerable to differential attacks and probably several other attacks as well. This is in clear contrast to the variants with 2×2 cipher block, which is completely in line with the full AES' square cipher block and has the same diffusion properties.

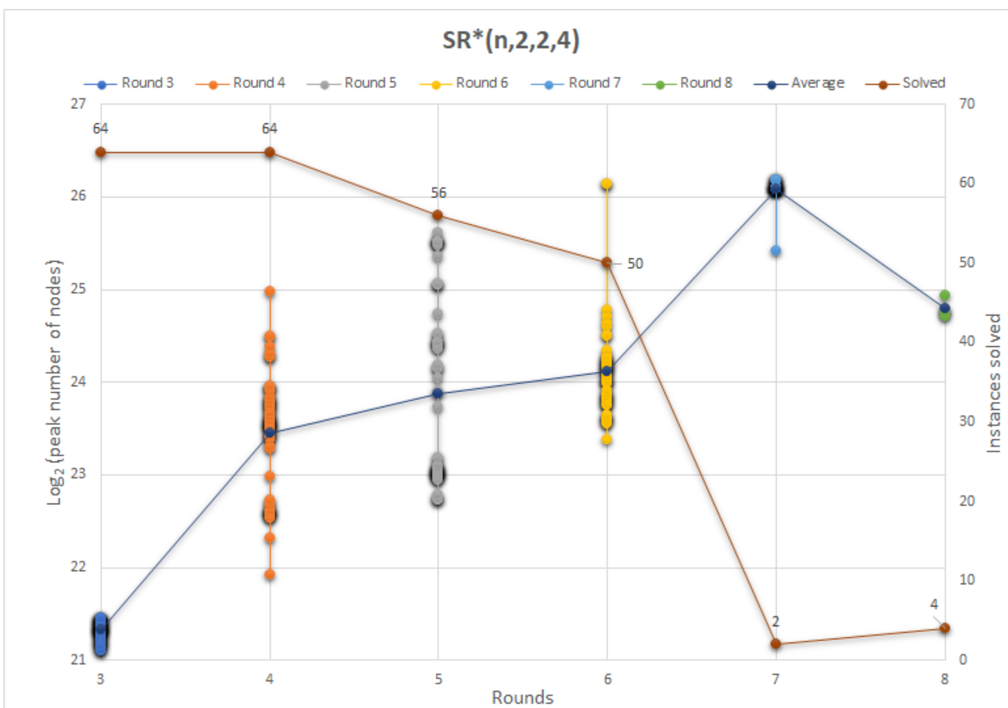
For the average time it takes to complete a run-through we observe the following: For systems that largely got solved, the time for run-throughs increases with the number of rounds, as expected. Another interesting observation is that while we were able to solve roughly the same fraction of instances both for 1×4 and 2×2 block sizes, it clearly takes longer time to solve the 2×2 instances, and it takes a little more memory as well. This can be seen in Figure 5.5. It appears that the 2×2 instances are harder to solve after all, as they consume more time and memory, but there is less difference between the 2×2 and 1×4 instances than there is between the 2×2 and 4×1 instances.

5.3.3 Indications of Plaintext Dependencies

Finally, we try to make some observations on which plaintext constants that may give easier systems to solve. In SR*(7,2,2,4) we solve two instances, while in SR*(8,2,2,4) we solve four. In Figure 5.6 we see that the four solved instances of SR*(8,2,2,4) all were based on the plaintext



(a) SR*$(n, 1, 4, 4)$

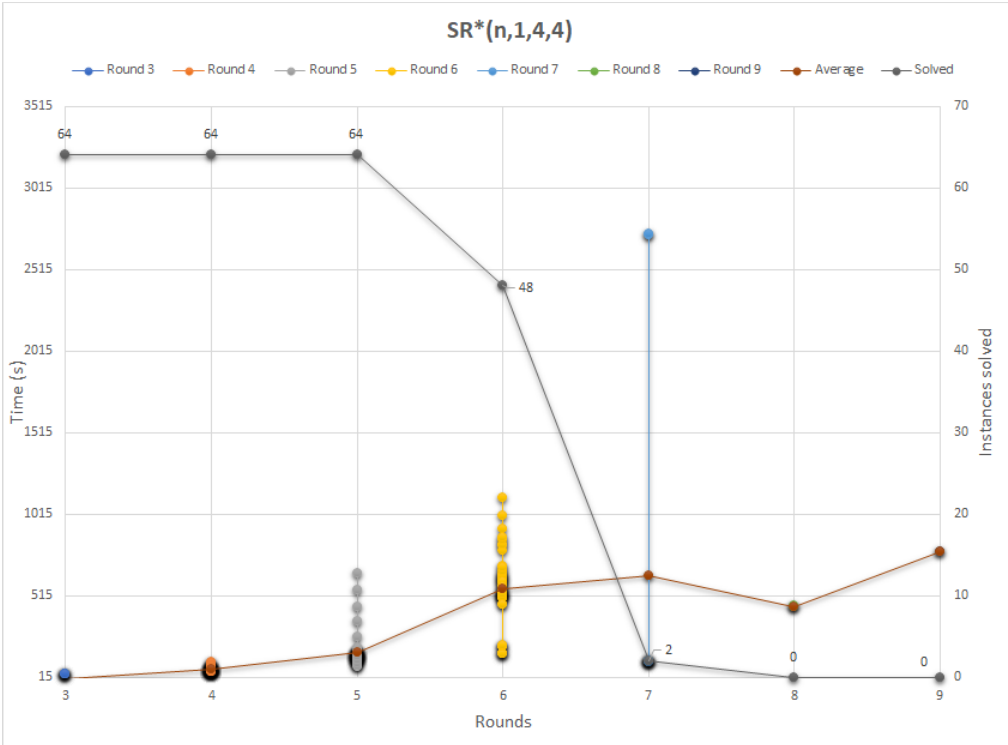


(b) SR*$(n, 2, 2, 4)$

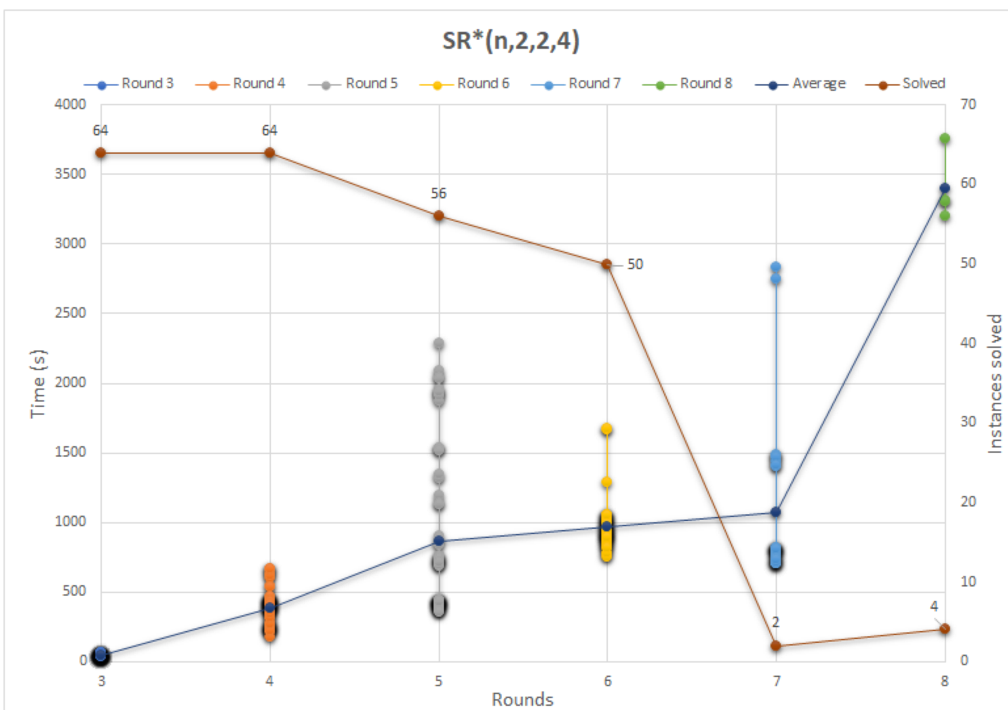
Figure 5.4: Average complexity with distribution, and number of instances solved.

with the hexadecimal value **ff00**, while the two systems solved in SR*(7,2,2,4) were based on different plaintexts. Looking into the nine instances of SR*(3,2,4,4) we were able to solve we find that four of them had the plaintext value **aaaaaaaa**. All of these were solved considerably faster than the other five.

These observations on plaintexts that seem to give systems that are easier to solve do not have a strong enough basis to draw any firm conclusions. But I think they indicate something that is worth to study further.



(a) SR*$(n, 1, 4, 4)$



(b) SR*$(n, 2, 2, 4)$

Figure 5.5: Average time with distribution, and number of instances solved.

Count of System	Column Labels															
Row Labels	SR*(3,1,4,4)	SR*(3,2,2,4)	SR*(3,4,1,4)	SR*(4,1,4,4)	SR*(4,2,2,4)	SR*(4,4,1,4)	SR*(5,1,4,4)	SR*(5,2,2,4)	SR*(6,1,4,4)	SR*(6,2,2,4)	SR*(7,1,4,4)	SR*(7,2,2,4)	SR*(8,1,4,4)	SR*(8,2,2,4)	SR*(9,1,4,4)	Grand Total
00ff	**8**	**8**		**8**	**8**		**8**	**8**	**2**	**2**		**1**				**53**
0										2						2
1	8	8		8	8		8	8	2			1				51
174c	**8**	**8**	**1**	**8**	**8**		**8**	**5**	**8**	**8**		**8**	**2**			**72**
0			1									8	2			11
1	8	8		8	8		8	5	8	8						61
5555	**8**	**8**		**8**	**8**		**8**	**7**	**8**	**8**						**63**
1	8	8		8	8		8	7	8	8						63
94b3	**8**	**8**		**8**	**8**		**8**	**5**	**7**	**8**	**1**	**6**				**67**
0												6				6
1	8	8		8	8		8	5	7	8	1					61
aaaa	**8**	**8**		**8**	**8**		**8**	**8**	**8**	**8**		**8**				**72**
0												8				8
1	8	8		8	8		8	8	8	8						64
bbff	**8**	**8**		**8**	**8**		**8**	**8**		**8**	**1**					**57**
1	8	8		8	8		8	8		8	1					57
dbc5	**8**	**8**		**8**	**8**		**8**	**7**	**7**	**8**		**1**				**63**
1	8	8		8	8		8	7	7	8		1				63
ff00	**8**	**8**		**8**	**8**	**5**	**8**	**8**	**8**	**2**	**8**	**8**		**4**	**1**	**84**
0						5					8	8			1	22
1	8	8		8	8		8	8	8	2				4		62
Grand Total	**64**	**64**	**1**	**64**	**64**	**5**	**64**	**56**	**48**	**52**	**10**	**32**	**2**	**4**	**1**	**531**

Figure 5.6: All non-lost instances and their spread among systems and plaintexts used for 16-bit instances.

Count of System	Column Labels														
Row Labels	*(3,2,4,4)	SR*(3,4,2,4)	SR*(4,2,4,4)	SR*(4,4,2,4)	SR*(5,2,4,4)	SR*(5,4,2,4)	SR*(6,2,4,4)	SR*(6,4,2,4)	SR*(7,2,4,4)	SR*(7,4,2,4)	SR*(8,2,4,4)	SR*(8,4,2,4)	SR*(9,2,4,4)	SR*(9,4,2,4)	Grand Total
0000ffff	8	8	8	5	8	8	8	6	8	7	6	8	5	5	98
0	7	8	8	5	8	8	8	6	8	7	6	8	5	5	97
1	1														1
174ca832	8	8	8	8	8	8	8	8	8	8	8	8	5	7	108
0	8	8	8	8	8	8	8	8	8	8	8	8	5	7	108
55555555	8	8	8	8	8	7	8	8	8	8	6	8		7	100
0	7	8	8	8	8	7	8	8	8	8	6	8		7	99
1	1														1
94b3de7f	8	8	8	8	8	8	8	8	6	8	7	8	8	7	108
0	6	8	8	8	8	8	8	8	6	8	7	8	8	7	106
1	2														2
aaaaaaaa	8	8	8	8	8	8	8	8	8	8	7	8	5	8	108
0	4	8	8	8	8	8	8	8	8	8	7	8	5	8	104
1	4														4
bbbbffff	8	8	8	8	8	8	8	8	4	8	7	8	6	8	105
0	7	8	8	8	8	8	8	8	4	8	7	8	6	8	104
1	1														1
dbc5a241	8	8	8	8	8	7	8	8	8	8	6	8	8	7	108
0	8	8	8	8	8	7	8	8	8	8	6	8	8	7	108
ffff0000	8	8	8	8	8	8	8	8	8	8	7	8	5	7	107
0	8	8	8	8	8	8	8	8	8	8	7	8	5	7	107
Grand Total	64	64	64	61	64	62	64	62	58	63	54	64	42	56	842

Figure 5.7: All non-lost instances and their spread among systems and plaintexts used for 32-bit instances.

Chapter 6

Conclusion and Further Works

In this thesis we have expanded on the work done in [3, 11, 12], by using Compressed Right-Hand Side equations to attack small-scale versions of AES. Through this we have gained insight on the behaviour of CRHS on small-scale AES, and through this hopefully on AES itself. We have seen that system complexity and execution time has grown for variants with 16-bit key size as the number of rounds grow (see Figure 6.1 and Figure 6.2). We observe differences in complexity within variants of 16 bits but with different state array dimensions. It is clear that a higher number of rows in the state array makes for higher complexity and execution time.

For the 32-bit variants, the hardware limit of 8 GB seems to limit the attacks in such a way that little useful information can be deduced. The significant exception to this is the nine solved run-throughs of SR*(3,2,4,4). These solved run-throughs had an average complexity of $\approx 2^{23,76}$ nodes, which is less than the 2^{32} complexity associated with brute-force. To the authors best knowledge this is the first time small-scale versions of 32-bit keys have been broken using MRHS or CRHS.

Furthermore, we have observed indications that complexity and execution time is dependent on the plaintext value used for the run-through. The cause of this is as of yet undetermined, and seems to be an interesting topic for further research.

Admittedly, the lost run-throughs skews the results. Although we cannot be sure, we believe that these lost run-thorugh would have affected the results in a "negative" way. By negative we mean that they probably would have led to a higher complexity and thus higher execution time. We base this on our belief that they were lost due to their large total size in the first place.

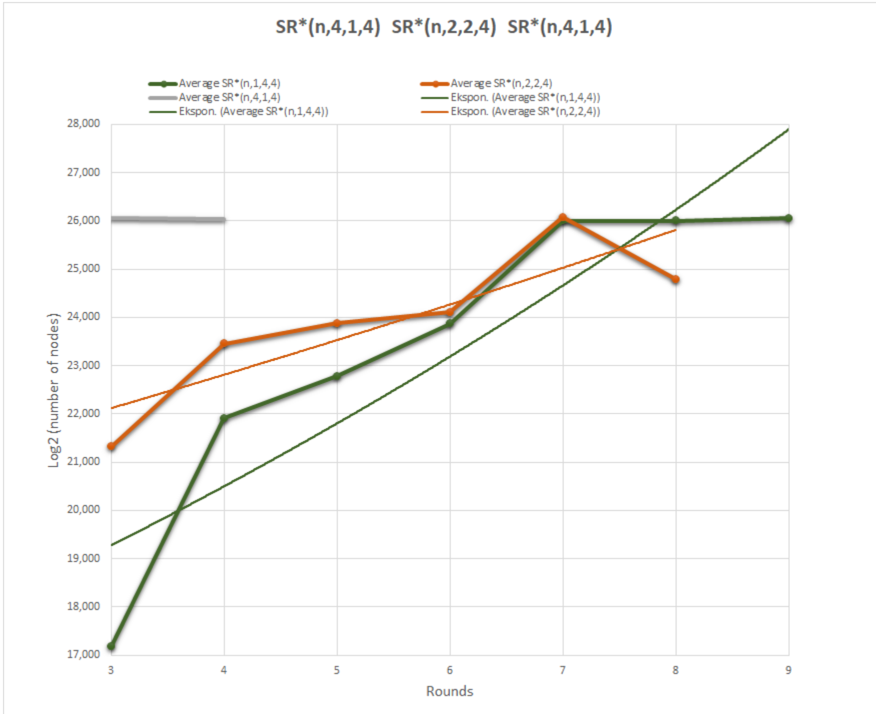


Figure 6.1: Average complexity per round for the 16-bit systems, with trend line.

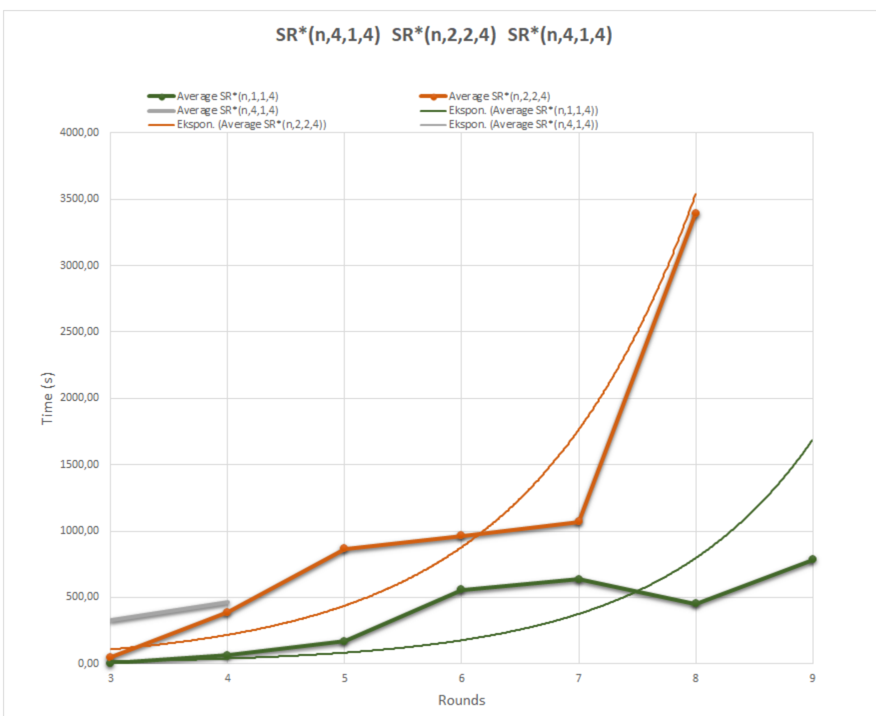


Figure 6.2: Average time per round for the 16-bit systems, with trend line.

The limited success on 32- and 64- bit size variants emphasizes the need for further improvements and refinements. In [11], cryptanalytic methods was used instead of automatic ordering to determine the order the BDDs should be joined in. They only attacked SR*$(n,2,2,4)$ but reports lower complexity for all rounds. This suggests that the automatic ordering strategy is sub-optimal. It would therefore be interesting to see what results would come of running these attacks again, with more emphasis on the ordering strategy. Work with this thesis has also resulted in discussions on how to improve the cryptanalysis strategy used in [11]. This has resultet in various ideas on possible improvement, though a fundamental question needs answering: Is it possible to mathematically determine the expected best order of joining BDDs, utilizing the fact that the initial linear combinations of a CRHS equation stay almost the same for every run-through?

Furthermore, the discovery of plaintext influence intrigues. Some plaintexts appears to give easier systems to solve than others. Can we explain this behavior and identify such plaintexts? Or is it simply that the variables makes for fewer/more nodes in the initial BDD, and thus "tricks" the automatic ordering into choosing a better/worse gluing order?

I am looking forward for the opportunity to study these and other questions as a PhD candidate!

Bibliography

[1] top500.org. `https://www.top500.org/list/2017/11/`, 2017. [Online: accessed 07-Feb-2018].

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677 – 691, 1986.

[3] C. Cid, S. Murphy, and M. Robshaw. Small scale variants of the aes. In *Fast Software Encryption - FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 145 – 162. Springer, 2005.

[4] J. B. Fraleigh. *A First Course in Abstract Algebra*. Pearson, seventh edition, 2003.

[5] D. Knuth. *Bitwise tricks and techniques; Binary Decision Diagrams*. Number Vol 4, Fascicle 1 in The Art of Computer Programming. ADDISON WESLEY (PEAR), 2009.

[6] D. C. Lay. *Linear Algebra and Its Applications*. Pearson Education and Greg Tobin, third edition, 2006.

[7] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. Discrete Mathematics and Its Applications. 1996.

[8] R. Munroe. "2018". `https://xkcd.com/1935/`, 2018. Licensed under Creative Commons Attribution-NonCommercial 2.5 License.

[9] N. I. of Standards and Technology. Advanced encryption standard, 2001. FIPS PUBS 197.

[10] C. Paar and J. Pelzl. *Understanding Cryptography*. Springer, 2010. 2nd Corrected printing.

[11] H. Raddum and O. Kazymyrov. Algebraic attacks using binary decision diagrams. In *International Conference on Cryptography and Information Security in the Balkans- BalkanCryptSec 2014*, volume 9024 of *Lecture Notes in Computer Science*, pages 40 – 54. Springer Verlag, 2015.

[12] H. Raddum and I. Semaev. Solving multiple right-hand sides linear equations. *Designs, Codes and Cryptography*, 49:147 – 168, 2008.

[13] V. Rijmen and J. Daemen. *The Design of Rijndael*. Springer-Verlag, 2001.

[14] T. E. Schilling and H. Raddum. Analysis of trivium using compressed right hand side equations. In *International Conference on Information Security and Cryptology - ICISC 2011*, volume 7259 of *Lecture Notes in Computer Science*, pages 18 – 32. Springer Verlag, 2011.

[15] T. E. Schilling and H. Raddum. Solving compressed right hand side equation systems with linear absorption. In *Sequences and Their Applications - SETA 2012*, volume 7280 of *Lecture Notes in Computer Science*, pages 291 – 302. Springer Verlag, 2012.

[16] B. Schneier. Schneier's law. `https://www.schneier.com/blog/archives/2011/04/schneiers_law.html`, 2011. [Online: accessed 07-Feb-2018].

Appendix A

Plaintext Values

The respective eight values were used both as plaintext and key during the creation of the various Small-Scale instances:

16-Bit Plaintext Values:

- bbff
- aaaa
- 5555
- ff00
- 00ff
- 174c
- 94b3
- dbc5

32-Bit Plaintext Values:

- bbbbffff
- aaaaaaaa
- 55555555
- ffff0000
- 0000ffff
- 174ca832
- 94b3de7f
- dbc5a241

64-Bit Plaintext Values:

- bbbbbbbbffffffff
- aaaaaaaaaaaaaaaa
- 5555555555555555
- ffffffff00000000
- 00000000ffffffff
- 174ca832174ca832
- 94b3de7f94b3de7f
- dbc5a241dbc5a241

Appendix B

Raw Data for 16 Bit Systems

Key size	System	ID	time in sec	# of nodes, as x in 2^x	1 if solved	plaintext	ciphertext	key
16bits	SR(9,1,4,4)	877036381.bdd	647	26,06	0	ff00	cf77	bbff
16bits	SRstar(3,1,4,4)	2056888506.bdd	0	17,158	1	dbc5	8ad1	00ff
16bits	SRstar(3,1,4,4)	1049840918.bdd	1	17,177	1	dbc5	3438	94b3
16bits	SRstar(3,1,4,4)	431695514.bdd	0	16,028	1	94b3	794d	00ff
16bits	SRstar(3,1,4,4)	-1979016437.bdd	0	17,286	1	5555	dc68	94b3
16bits	SRstar(3,1,4,4)	635808998.bdd	0	17,129	1	5555	8057	ff00
16bits	SRstar(3,1,4,4)	-341941834.bdd	0	17,083	1	94b3	450e	ff00
16bits	SRstar(3,1,4,4)	542282785.bdd	0	17,125	1	5555	1891	00ff
16bits	SRstar(3,1,4,4)	1285464618.bdd	0	15,685	1	bbff	1ea8	174c
16bits	SRstar(3,1,4,4)	1090601706.bdd	0	16,957	1	94b3	61a4	94b3
16bits	SRstar(3,1,4,4)	849160217.bdd	0	17,081	1	174c	a7fa	174c
16bits	SRstar(3,1,4,4)	-1576663552.bdd	0	17,185	1	aaaa	de43	aaaa
16bits	SRstar(3,1,4,4)	-354322275.bdd	0	17,123	1	5555	9353	174c
16bits	SRstar(3,1,4,4)	867059130.bdd	0	17,023	1	174c	c25a	00ff
16bits	SRstar(3,1,4,4)	1096668442.bdd	0	15,658	1	bbff	2aa8	00ff
16bits	SRstar(3,1,4,4)	-595509819.bdd	0	17,102	1	dbc5	7a89	bbff
16bits	SRstar(3,1,4,4)	1667731187.bdd	0	15,59	1	00ff	c5fa	aaaa
16bits	SRstar(3,1,4,4)	1117324306.bdd	0	17,09	1	dbc5	4e13	174c
16bits	SRstar(3,1,4,4)	-2097518206.bdd	0	17,15	1	94b3	e64b	174c
16bits	SRstar(3,1,4,4)	64064454.bdd	0	17,271	1	5555	161f	5555
16bits	SRstar(3,1,4,4)	-2125562522.bdd	0	17,246	1	aaaa	fd3d	174c
16bits	SRstar(3,1,4,4)	364003974.bdd	0	15,663	1	bbff	21fa	aaaa
16bits	SRstar(3,1,4,4)	-268649362.bdd	0	17,185	1	dbc5	7119	aaaa
16bits	SRstar(3,1,4,4)	-68429330.bdd	0	17,195	1	dbc5	6ed7	ff00
16bits	SRstar(3,1,4,4)	-1688136099.bdd	1	17,237	1	174c	7054	5555
16bits	SRstar(3,1,4,4)	1905174084.bdd	0	15,673	1	bbff	2a5b	bbff
16bits	SRstar(3,1,4,4)	599260613.bdd	0	17,186	1	174c	b662	aaaa
16bits	SRstar(3,1,4,4)	964348422.bdd	36	21,083	1	ff00	37b8	aaaa
16bits	SRstar(3,1,4,4)	-1305751417.bdd	1	17,123	1	aaaa	0867	dbc5
16bits	SRstar(3,1,4,4)	1197303449.bdd	1	17,254	1	aaaa	d7a5	5555
16bits	SRstar(3,1,4,4)	-1769401851.bdd	34	20,943	1	ff00	8d6e	5555
16bits	SRstar(3,1,4,4)	-567061330.bdd	1	17,23	1	aaaa	b02b	94b3
16bits	SRstar(3,1,4,4)	1443697430.bdd	2	17,615	1	ff00	606f	bbff
16bits	SRstar(3,1,4,4)	-2124400670.bdd	0	17,195	1	5555	fb79	aaaa
16bits	SRstar(3,1,4,4)	-504295813.bdd	0	15,629	1	00ff	0e51	94b3
16bits	SRstar(3,1,4,4)	1826310559.bdd	1	17,28	1	94b3	038b	aaaa
16bits	SRstar(3,1,4,4)	1535557939.bdd	36	21,002	1	ff00	c0c3	dbc5
16bits	SRstar(3,1,4,4)	-966862410.bdd	0	17,238	1	dbc5	921b	dbc5
16bits	SRstar(3,1,4,4)	231012934.bdd	0	17,065	1	94b3	61e0	5555

Figure B.1: Raw data for the 16 bit systems.

16bits	SRstar(3,1,4,4)	11296926.bdd	34	20,997	1	ff00	6019	00ff
16bits	SRstar(3,1,4,4)	898843927.bdd	37	21,066	1	ff00	6cd5	174c
16bits	SRstar(3,1,4,4)	1201304740.bdd	1	17,131	1	174c	42ba	94b3
16bits	SRstar(3,1,4,4)	-406423923.bdd	1	17,084	1	5555	98e9	bbff
16bits	SRstar(3,1,4,4)	434798140.bdd	0	17,116	1	174c	39f9	ff00
16bits	SRstar(3,1,4,4)	60792208.bdd	0	15,619	1	bbff	1e95	ff00
16bits	SRstar(3,1,4,4)	-1457576264.bdd	0	15,641	1	00ff	d5a8	00ff
16bits	SRstar(3,1,4,4)	405623711.bdd	0	17,048	1	174c	7d70	dbc5
16bits	SRstar(3,1,4,4)	-696340381.bdd	0	15,691	1	00ff	ef95	ff00
16bits	SRstar(3,1,4,4)	-1389792516.bdd	1	17,145	1	aaaa	d42f	ff00
16bits	SRstar(3,1,4,4)	-1080348685.bdd	0	17,268	1	aaaa	e464	00ff
16bits	SRstar(3,1,4,4)	-910503864.bdd	0	17,206	1	dbc5	aebf	5555
16bits	SRstar(3,1,4,4)	1428402096.bdd	0	17,179	1	94b3	1948	dbc5
16bits	SRstar(3,1,4,4)	-780003291.bdd	0	15,652	1	00ff	0b9b	5555
16bits	SRstar(3,1,4,4)	-1399816292.bdd	0	15,637	1	00ff	27b1	dbc5
16bits	SRstar(3,1,4,4)	-1895016868.bdd	0	16,054	1	94b3	e610	bbff
16bits	SRstar(3,1,4,4)	-1487146410.bdd	0	17,131	1	174c	bb98	bbff
16bits	SRstar(3,1,4,4)	1828226361.bdd	33	21,095	1	ff00	9f06	94b3
16bits	SRstar(3,1,4,4)	1444552798.bdd	36	21,018	1	ff00	ba14	ff00
16bits	SRstar(3,1,4,4)	1849245751.bdd	0	15,602	1	bbff	ee9b	5555
16bits	SRstar(3,1,4,4)	1191335875.bdd	0	17,129	1	aaaa	85aa	bbff
16bits	SRstar(3,1,4,4)	1424201236.bdd	0	15,624	1	bbff	8451	94b3
16bits	SRstar(3,1,4,4)	619189681.bdd	0	15,634	1	bbff	b2b1	dbc5
16bits	SRstar(3,1,4,4)	-1458417248.bdd	0	15,684	1	00ff	d75b	bbff
16bits	SRstar(3,1,4,4)	47842784.bdd	0	17,212	1	5555	de3b	dbc5
16bits	SRstar(3,1,4,4)	1094615662.bdd	0	15,651	1	00ff	d4a8	174c
16bits	SRstar(3,2,2,4)	1648748098.bdd	48	21,336	1	aaaa	e289	dbc5
16bits	SRstar(3,2,2,4)	92924584.bdd	49	21,349	1	94b3	870a	94b3
16bits	SRstar(3,2,2,4)	494564095.bdd	50	21,436	1	bbff	b685	dbc5
16bits	SRstar(3,2,2,4)	1561742825.bdd	67	21,199	1	00ff	baa9	bbff
16bits	SRstar(3,2,2,4)	386387000.bdd	52	21,407	1	dbc5	9ce3	bbff
16bits	SRstar(3,2,2,4)	1674629214.bdd	47	21,332	1	aaaa	0013	ff00
16bits	SRstar(3,2,2,4)	-926554255.bdd	42	21,196	1	bbff	bfcc	174c
16bits	SRstar(3,2,2,4)	345458712.bdd	41	21,219	1	dbc5	afc6	00ff
16bits	SRstar(3,2,2,4)	410819764.bdd	46	21,332	1	bbff	ac8b	00ff
16bits	SRstar(3,2,2,4)	795322177.bdd	45	21,273	1	aaaa	2fe0	5555
16bits	SRstar(3,2,2,4)	755322870.bdd	61	21,161	1	00ff	f1a5	174c
16bits	SRstar(3,2,2,4)	-496557060.bdd	63	21,105	1	00ff	39f0	94b3
16bits	SRstar(3,2,2,4)	1911896056.bdd	61	21,249	1	00ff	d90c	5555
16bits	SRstar(3,2,2,4)	-2017578305.bdd	46	21,313	1	174c	1234	94b3
16bits	SRstar(3,2,2,4)	1526096063.bdd	47	21,428	1	ff00	32c1	94b3
16bits	SRstar(3,2,2,4)	-575670107.bdd	48	21,418	1	174c	8877	dbc5
16bits	SRstar(3,2,2,4)	-178908270.bdd	37	21,109	1	ff00	565b	bbff
16bits	SRstar(3,2,2,4)	52196201.bdd	62	21,159	1	00ff	670f	ff00
16bits	SRstar(3,2,2,4)	1502524563.bdd	62	21,178	1	00ff	d2dc	00ff
16bits	SRstar(3,2,2,4)	1329197860.bdd	47	21,332	1	dbc5	0925	aaaa
16bits	SRstar(3,2,2,4)	-1359039206.bdd	49	21,343	1	bbff	0611	ff00
16bits	SRstar(3,2,2,4)	-828327307.bdd	47	21,393	1	ff00	1d91	5555
16bits	SRstar(3,2,2,4)	-1861400.bdd	46	21,447	1	bbff	18a2	bbff
16bits	SRstar(3,2,2,4)	1251261754.bdd	42	21,274	1	ff00	4747	00ff
16bits	SRstar(3,2,2,4)	-2023901517.bdd	47	21,37	1	174c	0b2b	aaaa
16bits	SRstar(3,2,2,4)	77806265.bdd	49	21,352	1	bbff	ed9a	5555
16bits	SRstar(3,2,2,4)	429135174.bdd	45	21,295	1	174c	95b5	ff00
16bits	SRstar(3,2,2,4)	2044897112.bdd	48	21,417	1	aaaa	539c	aaaa
16bits	SRstar(3,2,2,4)	116578515.bdd	45	21,308	1	bbff	9dbd	aaaa
16bits	SRstar(3,2,2,4)	-91390162.bdd	53	21,336	1	94b3	98ed	ff00
16bits	SRstar(3,2,2,4)	-9651044.bdd	49	21,337	1	174c	b869	bbff
16bits	SRstar(3,2,2,4)	1582198581.bdd	47	21,354	1	5555	60ab	dbc5
16bits	SRstar(3,2,2,4)	416054077.bdd	49	21,434	1	94b3	56c1	00ff
16bits	SRstar(3,2,2,4)	-1248524974.bdd	46	21,331	1	5555	513e	5555
16bits	SRstar(3,2,2,4)	-223282925.bdd	49	21,432	1	5555	d2c7	174c
16bits	SRstar(3,2,2,4)	-1261195292.bdd	51	21,401	1	5555	7300	ff00
16bits	SRstar(3,2,2,4)	1071353089.bdd	47	21,325	1	aaaa	feee	00ff
16bits	SRstar(3,2,2,4)	-232322026.bdd	45	21,406	1	ff00	a935	dbc5
16bits	SRstar(3,2,2,4)	562407091.bdd	47	21,317	1	aaaa	b45e	174c
16bits	SRstar(3,2,2,4)	2071172912.bdd	50	21,349	1	174c	a2d8	00ff
16bits	SRstar(3,2,2,4)	195081227.bdd	49	21,438	1	94b3	5c14	dbc5
16bits	SRstar(3,2,2,4)	1147944783.bdd	61	21,227	1	00ff	5c0c	dbc5
16bits	SRstar(3,2,2,4)	308993133.bdd	50	21,351	1	5555	c6bb	00ff
16bits	SRstar(3,2,2,4)	-1595153385.bdd	40	21,179	1	aaaa	0f1d	bbff
16bits	SRstar(3,2,2,4)	1032936311.bdd	48	21,45	1	5555	8dda	bbff
16bits	SRstar(3,2,2,4)	1757559886.bdd	47	21,328	1	94b3	6873	bbff
16bits	SRstar(3,2,2,4)	78540416.bdd	46	21,281	1	aaaa	765b	94b3

Figure B.2: Raw data for the 16 bit systems.

16bits	SRstar(3,2,2,4)	1078283706.bdd	46	21,449	1	ff00	8a74	ff00
16bits	SRstar(3,2,2,4)	-77957534.bdd	41	21,203	1	bbff	fef1	94b3
16bits	SRstar(3,2,2,4)	2052666451.bdd	47	21,315	1	dbc5	6dbf	174c
16bits	SRstar(3,2,2,4)	855011747.bdd	43	21,263	1	5555	9996	aaaa
16bits	SRstar(3,2,2,4)	1541148326.bdd	46	21,114	1	94b3	9f8f	174c
16bits	SRstar(3,2,2,4)	-2088886242.bdd	49	21,416	1	94b3	69f1	aaaa
16bits	SRstar(3,2,2,4)	-1464510278.bdd	47	21,429	1	174c	f1af	174c
16bits	SRstar(3,2,2,4)	-696811087.bdd	40	21,227	1	174c	7fb7	5555
16bits	SRstar(3,2,2,4)	2078810294.bdd	48	21,439	1	dbc5	e35a	ff00
16bits	SRstar(3,2,2,4)	742048230.bdd	46	21,333	1	dbc5	e803	94b3
16bits	SRstar(3,2,2,4)	-1881055423.bdd	46	21,328	1	5555	fdd1	94b3
16bits	SRstar(3,2,2,4)	298560859.bdd	48	21,441	1	94b3	84c8	5555
16bits	SRstar(3,2,2,4)	-748261822.bdd	46	21,309	1	ff00	022c	174c
16bits	SRstar(3,2,2,4)	-2011408438.bdd	63	21,164	1	00ff	04c5	aaaa
16bits	SRstar(3,2,2,4)	-80725789.bdd	47	21,361	1	dbc5	6b01	5555
16bits	SRstar(3,2,2,4)	-1987357920.bdd	48	21,453	1	dbc5	cb78	dbc5
16bits	SRstar(3,2,2,4)	1788122704.bdd	45	21,386	1	ff00	d213	aaaa
16bits	SRstar(3,4,1,4)	273106702.bdd	330	26,06	0	174c	2def	94b3
16bits	SRstar(4,1,4,4)	-1656176334.bdd	60	21,925	1	94b3	0e49	ff00
16bits	SRstar(4,1,4,4)	665951563.bdd	59	21,246	1	00ff	980f	00ff
16bits	SRstar(4,1,4,4)	268309590.bdd	84	21,894	1	ff00	ef9b	ff00
16bits	SRstar(4,1,4,4)	1662921630.bdd	67	22,1	1	174c	f19e	bbff
16bits	SRstar(4,1,4,4)	-1116145719.bdd	52	21,658	1	dbc5	19ed	00ff
16bits	SRstar(4,1,4,4)	506164903.bdd	56	21,191	1	bbff	12bf	94b3
16bits	SRstar(4,1,4,4)	-2036053563.bdd	64	21,844	1	174c	bc5d	5555
16bits	SRstar(4,1,4,4)	1214783412.bdd	69	22,023	1	dbc5	ea77	5555
16bits	SRstar(4,1,4,4)	-941927293.bdd	48	21,894	1	bbff	62ab	bbff
16bits	SRstar(4,1,4,4)	-1789773861.bdd	70	22,137	1	174c	5c78	174c
16bits	SRstar(4,1,4,4)	363718995.bdd	59	21,933	1	94b3	a1db	174c
16bits	SRstar(4,1,4,4)	-363439164.bdd	99	21,95	1	ff00	6ddb	dbc5
16bits	SRstar(4,1,4,4)	-212247801.bdd	60	21,93	1	94b3	fba6	94b3
16bits	SRstar(4,1,4,4)	365360100.bdd	56	21,665	1	00ff	7663	5555
16bits	SRstar(4,1,4,4)	-2003238308.bdd	72	22,056	1	aaaa	27b6	174c
16bits	SRstar(4,1,4,4)	425730888.bdd	58	21,916	1	aaaa	8484	00ff
16bits	SRstar(4,1,4,4)	1660040060.bdd	63	22,016	1	5555	b6c3	ff00
16bits	SRstar(4,1,4,4)	623202040.bdd	68	21,892	1	aaaa	a3b8	bbff
16bits	SRstar(4,1,4,4)	-1184431395.bdd	62	21,181	1	00ff	e0bf	94b3
16bits	SRstar(4,1,4,4)	31477427.bdd	65	22,152	1	94b3	3af1	bbff
16bits	SRstar(4,1,4,4)	-297671432.bdd	60	22,139	1	94b3	e6c6	dbc5
16bits	SRstar(4,1,4,4)	797740944.bdd	69	22,026	1	5555	4ba3	dbc5
16bits	SRstar(4,1,4,4)	-16592029.bdd	59	21,93	1	aaaa	0041	5555
16bits	SRstar(4,1,4,4)	-1983702523.bdd	67	21,969	1	174c	fa49	dbc5
16bits	SRstar(4,1,4,4)	704255381.bdd	56	21,26	1	bbff	d90f	00ff
16bits	SRstar(4,1,4,4)	-399157084.bdd	84	22,031	1	5555	e44a	bbff
16bits	SRstar(4,1,4,4)	1317463770.bdd	59	21,909	1	dbc5	92da	bbff
16bits	SRstar(4,1,4,4)	537277672.bdd	54	21,79	1	dbc5	8643	174c
16bits	SRstar(4,1,4,4)	-659786186.bdd	60	21,992	1	174c	5fc1	aaaa
16bits	SRstar(4,1,4,4)	1164861678.bdd	58	21,905	1	00ff	b40e	174c
16bits	SRstar(4,1,4,4)	-1100683743.bdd	55	21,808	1	bbff	c33b	aaaa
16bits	SRstar(4,1,4,4)	-1468911607.bdd	66	21,929	1	94b3	fc28	aaaa
16bits	SRstar(4,1,4,4)	1797440752.bdd	87	21,855	1	ff00	d5e9	174c
16bits	SRstar(4,1,4,4)	404412836.bdd	61	22,019	1	174c	a59b	94b3
16bits	SRstar(4,1,4,4)	-1088133419.bdd	57	21,802	1	dbc5	5033	ff00
16bits	SRstar(4,1,4,4)	184869129.bdd	97	21,938	1	ff00	5833	94b3
16bits	SRstar(4,1,4,4)	834776226.bdd	59	22,029	1	bbff	9e24	dbc5
16bits	SRstar(4,1,4,4)	710682983.bdd	65	21,88	1	aaaa	3271	ff00
16bits	SRstar(4,1,4,4)	485394522.bdd	56	21,77	1	dbc5	c21d	94b3
16bits	SRstar(4,1,4,4)	1976972169.bdd	60	22,164	1	94b3	2bd7	00ff
16bits	SRstar(4,1,4,4)	1946609432.bdd	56	22,056	1	bbff	90f7	ff00
16bits	SRstar(4,1,4,4)	-418468599.bdd	58	21,74	1	aaaa	8890	aaaa
16bits	SRstar(4,1,4,4)	-1694822187.bdd	69	21,945	1	aaaa	3295	dbc5
16bits	SRstar(4,1,4,4)	2121131546.bdd	67	22,086	1	94b3	61b9	5555
16bits	SRstar(4,1,4,4)	783548941.bdd	61	21,926	1	00ff	0124	dbc5
16bits	SRstar(4,1,4,4)	-139038942.bdd	66	22,065	1	aaaa	4608	94b3
16bits	SRstar(4,1,4,4)	-2032238408.bdd	65	22,166	1	5555	7213	174c
16bits	SRstar(4,1,4,4)	948105556.bdd	68	22,029	1	174c	6da4	ff00
16bits	SRstar(4,1,4,4)	1734050549.bdd	60	21,926	1	dbc5	8e53	dbc5
16bits	SRstar(4,1,4,4)	-657701153.bdd	96	21,919	1	ff00	4b3f	bbff
16bits	SRstar(4,1,4,4)	-670233829.bdd	71	22,166	1	5555	3f2d	00ff
16bits	SRstar(4,1,4,4)	-884551014.bdd	73	21,946	1	00ff	27ab	bbff
16bits	SRstar(4,1,4,4)	-73961403.bdd	49	21,922	1	ff00	d27d	aaaa
16bits	SRstar(4,1,4,4)	-413546219.bdd	65	21,923	1	5555	1419	aaaa
16bits	SRstar(4,1,4,4)	446097532.bdd	58	21,918	1	00ff	a63b	aaaa

Figure B.3: Raw data for the 16 bit systems.

16bits	SRstar(4,1,4,4)	-277856836.bdd	59	21,863	1	174c	b319	00ff
16bits	SRstar(4,1,4,4)	-1613768096.bdd	62	21,918	1	5555	932d	94b3
16bits	SRstar(4,1,4,4)	-1041790573.bdd	54	22,076	1	bbff	1a63	5555
16bits	SRstar(4,1,4,4)	-1305832440.bdd	112	21,909	1	ff00	f04b	00ff
16bits	SRstar(4,1,4,4)	1728061883.bdd	58	22,083	1	bbff	160e	174c
16bits	SRstar(4,1,4,4)	-167613546.bdd	82	22,109	1	5555	ad07	5555
16bits	SRstar(4,1,4,4)	351533907.bdd	70	21,882	1	ff00	8bcf	5555
16bits	SRstar(4,1,4,4)	-230121783.bdd	64	21,999	1	00ff	28f7	ff00
16bits	SRstar(4,1,4,4)	969350887.bdd	59	21,928	1	dbc5	3309	aaaa
16bits	SRstar(4,2,2,4)	-541968475.bdd	236	22,615	1	ff00	2bf1	ff00
16bits	SRstar(4,2,2,4)	-165296167.bdd	366	23,587	1	bbff	ad70	174c
16bits	SRstar(4,2,2,4)	-1653910013.bdd	602	24,268	1	00ff	7301	ff00
16bits	SRstar(4,2,2,4)	1785050567.bdd	316	23,38	1	ff00	5f0d	94b3
16bits	SRstar(4,2,2,4)	-283259851.bdd	420	23,673	1	174c	7cbf	dbc5
16bits	SRstar(4,2,2,4)	505735428.bdd	465	23,739	1	bbff	8521	aaaa
16bits	SRstar(4,2,2,4)	655265440.bdd	226	22,587	1	bbff	8fc5	5555
16bits	SRstar(4,2,2,4)	-2071017763.bdd	310	23,286	1	5555	99f9	dbc5
16bits	SRstar(4,2,2,4)	-2092771425.bdd	395	23,517	1	aaaa	2ebe	174c
16bits	SRstar(4,2,2,4)	556448478.bdd	431	23,937	1	bbff	67d5	dbc5
16bits	SRstar(4,2,2,4)	64340202.bdd	251	22,703	1	5555	b56e	94b3
16bits	SRstar(4,2,2,4)	958845665.bdd	426	23,798	1	5555	df2d	174c
16bits	SRstar(4,2,2,4)	-603689257.bdd	440	23,759	1	dbc5	b374	174c
16bits	SRstar(4,2,2,4)	-1782447345.bdd	438	23,804	1	dbc5	0bdc	dbc5
16bits	SRstar(4,2,2,4)	1510680003.bdd	377	23,414	1	5555	7a3f	5555
16bits	SRstar(4,2,2,4)	1023110422.bdd	384	23,845	1	ff00	91b4	bbff
16bits	SRstar(4,2,2,4)	122052114.bdd	222	22,325	1	aaaa	00ca	dbc5
16bits	SRstar(4,2,2,4)	526668868.bdd	390	23,534	1	aaaa	9797	bbff
16bits	SRstar(4,2,2,4)	140021630.bdd	363	23,75	1	94b3	993a	ff00
16bits	SRstar(4,2,2,4)	-597397680.bdd	248	22,527	1	ff00	971c	aaaa
16bits	SRstar(4,2,2,4)	349058484.bdd	438	23,936	1	174c	ab99	ff00
16bits	SRstar(4,2,2,4)	899134970.bdd	433	23,558	1	174c	222b	174c
16bits	SRstar(4,2,2,4)	-2083609214.bdd	218	22,571	1	dbc5	212a	aaaa
16bits	SRstar(4,2,2,4)	-1086726051.bdd	458	23,714	1	ff00	87e9	dbc5
16bits	SRstar(4,2,2,4)	-75136493.bdd	270	22,983	1	94b3	50ca	bbff
16bits	SRstar(4,2,2,4)	1407625764.bdd	179	21,917	1	174c	e7ed	00ff
16bits	SRstar(4,2,2,4)	-526464191.bdd	436	23,595	1	aaaa	6da2	ff00
16bits	SRstar(4,2,2,4)	565341797.bdd	626	24,271	1	00ff	6d9f	5555
16bits	SRstar(4,2,2,4)	1125209898.bdd	384	23,523	1	174c	b9a5	5555
16bits	SRstar(4,2,2,4)	1853497361.bdd	233	22,597	1	5555	f4de	ff00
16bits	SRstar(4,2,2,4)	563616064.bdd	662	24,492	1	00ff	248f	dbc5
16bits	SRstar(4,2,2,4)	-1669765818.bdd	417	23,413	1	aaaa	f04d	5555
16bits	SRstar(4,2,2,4)	-1913939042.bdd	434	23,949	1	aaaa	c78f	aaaa
16bits	SRstar(4,2,2,4)	104796012.bdd	240	22,615	1	bbff	29f3	ff00
16bits	SRstar(4,2,2,4)	92923975.bdd	418	23,579	1	ff00	7088	5555
16bits	SRstar(4,2,2,4)	-689073906.bdd	271	22,604	1	ff00	276c	00ff
16bits	SRstar(4,2,2,4)	122671592.bdd	432	23,799	1	dbc5	825c	00ff
16bits	SRstar(4,2,2,4)	5620768.bdd	222	22,568	1	bbff	cac1	94b3
16bits	SRstar(4,2,2,4)	1526386271.bdd	381	23,504	1	dbc5	dc0b	5555
16bits	SRstar(4,2,2,4)	-990996377.bdd	381	23,521	1	94b3	39a5	174c
16bits	SRstar(4,2,2,4)	-2057316225.bdd	618	24,317	1	00ff	6458	00ff
16bits	SRstar(4,2,2,4)	-2012960149.bdd	252	22,582	1	5555	d08a	aaaa
16bits	SRstar(4,2,2,4)	696578144.bdd	419	23,621	1	ff00	9581	174c
16bits	SRstar(4,2,2,4)	1905536864.bdd	668	24,485	1	00ff	ff99	aaaa
16bits	SRstar(4,2,2,4)	-1644518572.bdd	657	24,383	1	00ff	e22f	174c
16bits	SRstar(4,2,2,4)	-635318265.bdd	385	23,517	1	5555	3084	bbff
16bits	SRstar(4,2,2,4)	-554433499.bdd	268	22,55	1	174c	1f94	bbff
16bits	SRstar(4,2,2,4)	-690887564.bdd	387	23,541	1	bbff	7a46	00ff
16bits	SRstar(4,2,2,4)	798420386.bdd	416	23,406	1	94b3	fa7f	00ff
16bits	SRstar(4,2,2,4)	-347316852.bdd	245	22,674	1	dbc5	d8b2	bbff
16bits	SRstar(4,2,2,4)	-1736149301.bdd	371	23,426	1	aaaa	7f28	00ff
16bits	SRstar(4,2,2,4)	2043552427.bdd	412	23,538	1	94b3	1b5e	5555
16bits	SRstar(4,2,2,4)	486927984.bdd	336	23,316	1	5555	87f7	00ff
16bits	SRstar(4,2,2,4)	-1110471156.bdd	392	23,576	1	94b3	7db5	aaaa
16bits	SRstar(4,2,2,4)	-1428461840.bdd	543	23,802	1	00ff	9973	94b3
16bits	SRstar(4,2,2,4)	523785363.bdd	224	22,726	1	dbc5	9952	ff00
16bits	SRstar(4,2,2,4)	479658979.bdd	409	23,962	1	174c	cbd7	aaaa
16bits	SRstar(4,2,2,4)	-1645304844.bdd	421	23,736	1	174c	305e	94b3
16bits	SRstar(4,2,2,4)	-1943104138.bdd	451	23,662	1	94b3	ad43	94b3
16bits	SRstar(4,2,2,4)	-1121874403.bdd	443	23,905	1	aaaa	e048	94b3
16bits	SRstar(4,2,2,4)	-2000334373.bdd	350	23,631	1	dbc5	b0b5	94b3
16bits	SRstar(4,2,2,4)	764617991.bdd	452	23,573	1	bbff	83ec	bbff
16bits	SRstar(4,2,2,4)	-1362638626.bdd	380	23,49	1	94b3	2484	dbc5
16bits	SRstar(4,2,2,4)	-1138817783.bdd	530	24,978	1	00ff	0ece	bbff

Figure B.4: Raw data for the 16 bit systems.

16bits	SRstar(4,4,1,4)	1619097554.bdd	203	26,029	0	ff00	21c2	00ff
16bits	SRstar(4,4,1,4)	1555130172.bdd	555	26,031	0	ff00	ca39	bbff
16bits	SRstar(4,4,1,4)	-354843854.bdd	563	26,032	0	ff00	fc85	94b3
16bits	SRstar(4,4,1,4)	-718402091.bdd	510	26,03	0	ff00	eb6d	aaaa
16bits	SRstar(4,4,1,4)	-654635616.bdd	512	26,032	0	ff00	0cce	174c
16bits	SRstar(5,1,4,4)	821791648.bdd	123	22,386	1	00ff	fe56	aaaa
16bits	SRstar(5,1,4,4)	390666478.bdd	116	22,938	1	00ff	0c54	00ff
16bits	SRstar(5,1,4,4)	-400870146.bdd	172	22,824	1	94b3	b749	94b3
16bits	SRstar(5,1,4,4)	892911082.bdd	118	22,346	1	00ff	4c90	dbc5
16bits	SRstar(5,1,4,4)	590970524.bdd	115	22,895	1	174c	1181	00ff
16bits	SRstar(5,1,4,4)	-1398083709.bdd	266	22,768	1	ff00	d2fa	94b3
16bits	SRstar(5,1,4,4)	905627106.bdd	128	22,485	1	00ff	d1b3	5555
16bits	SRstar(5,1,4,4)	-1329961330.bdd	201	22,304	1	ff00	56de	dbc5
16bits	SRstar(5,1,4,4)	-520006932.bdd	123	22,98	1	bbff	c854	00ff
16bits	SRstar(5,1,4,4)	1742291917.bdd	168	22,488	1	00ff	a455	ff00
16bits	SRstar(5,1,4,4)	-1784143996.bdd	158	22,744	1	94b3	8a43	aaaa
16bits	SRstar(5,1,4,4)	1455256898.bdd	164	22,744	1	174c	61e2	174c
16bits	SRstar(5,1,4,4)	1540162404.bdd	450	23,133	1	ff00	eaff	5555
16bits	SRstar(5,1,4,4)	1648015291.bdd	147	22,666	1	5555	0b36	dbc5
16bits	SRstar(5,1,4,4)	-1935173392.bdd	133	22,548	1	bbff	2b55	ff00
16bits	SRstar(5,1,4,4)	-285435042.bdd	160	22,706	1	174c	b526	5555
16bits	SRstar(5,1,4,4)	-1456349124.bdd	77	22,573	1	00ff	3d76	94b3
16bits	SRstar(5,1,4,4)	46881800.bdd	126	22,345	1	bbff	f790	dbc5
16bits	SRstar(5,1,4,4)	-207642064.bdd	143	22,763	1	5555	27ff	174c
16bits	SRstar(5,1,4,4)	520557088.bdd	137	22,668	1	dbc5	4d77	aaaa
16bits	SRstar(5,1,4,4)	-626359478.bdd	148	23,391	1	5555	bba7	aaaa
16bits	SRstar(5,1,4,4)	-15372190.bdd	171	22,866	1	aaaa	621c	dbc5
16bits	SRstar(5,1,4,4)	1849174701.bdd	148	22,665	1	174c	a8ef	dbc5
16bits	SRstar(5,1,4,4)	30922365.bdd	154	22,779	1	94b3	7957	174c
16bits	SRstar(5,1,4,4)	-1839819857.bdd	143	22,704	1	94b3	20f5	dbc5
16bits	SRstar(5,1,4,4)	-228530226.bdd	128	22,466	1	bbff	3d56	aaaa
16bits	SRstar(5,1,4,4)	-1798414504.bdd	150	22,641	1	174c	8c2d	ff00
16bits	SRstar(5,1,4,4)	1632840862.bdd	149	22,665	1	5555	87ad	5555
16bits	SRstar(5,1,4,4)	1091999081.bdd	145	22,698	1	aaaa	90cc	aaaa
16bits	SRstar(5,1,4,4)	-546639504.bdd	143	22,714	1	aaaa	5d89	bbff
16bits	SRstar(5,1,4,4)	-1940537469.bdd	126	22,446	1	bbff	352b	174c
16bits	SRstar(5,1,4,4)	-2143043790.bdd	137	22,461	1	00ff	f3bc	bbff
16bits	SRstar(5,1,4,4)	1988854055.bdd	156	22,718	1	dbc5	b726	dbc5
16bits	SRstar(5,1,4,4)	145418467.bdd	146	22,73	1	aaaa	10ec	5555
16bits	SRstar(5,1,4,4)	-682483283.bdd	157	22,812	1	aaaa	b874	ff00
16bits	SRstar(5,1,4,4)	-1510374745.bdd	177	23,442	1	94b3	0d0a	5555
16bits	SRstar(5,1,4,4)	961951772.bdd	158	22,744	1	94b3	e6cb	bbff
16bits	SRstar(5,1,4,4)	-99122853.bdd	166	22,145	1	ff00	8508	bbff
16bits	SRstar(5,1,4,4)	-1800414557.bdd	362	24,77	1	ff00	5310	00ff
16bits	SRstar(5,1,4,4)	1386738526.bdd	146	23,402	1	174c	2290	bbff
16bits	SRstar(5,1,4,4)	149573257.bdd	144	22,142	1	ff00	9c46	174c
16bits	SRstar(5,1,4,4)	-172739961.bdd	160	22,754	1	94b3	8926	00ff
16bits	SRstar(5,1,4,4)	-1454252448.bdd	89	22,628	1	bbff	8e76	94b3
16bits	SRstar(5,1,4,4)	982332903.bdd	93	22,608	1	aaaa	1cd7	94b3
16bits	SRstar(5,1,4,4)	-1276223921.bdd	134	22,583	1	5555	ab6f	bbff
16bits	SRstar(5,1,4,4)	-1261100619.bdd	162	22,746	1	94b3	9df2	ff00
16bits	SRstar(5,1,4,4)	1521901098.bdd	167	22,625	1	5555	efef	94b3
16bits	SRstar(5,1,4,4)	64070858.bdd	164	22,812	1	174c	e121	aaaa
16bits	SRstar(5,1,4,4)	-865148487.bdd	159	22,769	1	aaaa	47ae	00ff
16bits	SRstar(5,1,4,4)	-2133785760.bdd	121	23,186	1	dbc5	796d	5555
16bits	SRstar(5,1,4,4)	1463064437.bdd	151	22,771	1	5555	3aeb	ff00
16bits	SRstar(5,1,4,4)	-1817680871.bdd	130	22,438	1	bbff	c9b3	5555
16bits	SRstar(5,1,4,4)	-436837571.bdd	155	22,743	1	dbc5	683d	00ff
16bits	SRstar(5,1,4,4)	1434335335.bdd	101	23,044	1	5555	9d6d	00ff
16bits	SRstar(5,1,4,4)	330027409.bdd	161	22,715	1	dbc5	0e0f	94b3
16bits	SRstar(5,1,4,4)	1168545558.bdd	150	22,723	1	dbc5	1b9b	ff00
16bits	SRstar(5,1,4,4)	1342728211.bdd	153	22,708	1	174c	c562	94b3
16bits	SRstar(5,1,4,4)	1716234217.bdd	120	22,347	1	00ff	402b	174c
16bits	SRstar(5,1,4,4)	-2065942346.bdd	162	22,857	1	aaaa	d88c	174c
16bits	SRstar(5,1,4,4)	1515911603.bdd	155	22,751	1	dbc5	7c5f	bbff
16bits	SRstar(5,1,4,4)	1796840753.bdd	144	22,655	1	dbc5	156f	174c
16bits	SRstar(5,1,4,4)	-2065960640.bdd	127	22,371	1	bbff	dcbc	bbff
16bits	SRstar(5,1,4,4)	860957806.bdd	554	24,086	1	ff00	1cbb	aaaa
16bits	SRstar(5,1,4,4)	-1780674002.bdd	653	24,209	1	ff00	f503	ff00
16bits	SRstar(5,2,2,4)	-937744961.bdd	419	23,005	1	174c	d0a7	94b3
16bits	SRstar(5,2,2,4)	1872773162.bdd	1936	25,617	1	bbff	dfb6	5555
16bits	SRstar(5,2,2,4)	473306293.bdd	397	22,731	1	174c	3e5c	aaaa
16bits	SRstar(5,2,2,4)	-85399513.bdd	721	24,416	1	00ff	20fa	bbff

Figure B.5: Raw data for the 16 bit systems.

16bits	SRstar(5,2,2,4)	-1837371527.bdd	2086	25,36	1	94b3	9976	94b3
16bits	SRstar(5,2,2,4)	791834372.bdd	434	23,049	1	bbff	f750	dbc5
16bits	SRstar(5,2,2,4)	1688776110.bdd	1313	24,527	1	ff00	b70b	174c
16bits	SRstar(5,2,2,4)	-475144831.bdd	1516	25,058	1	ff00	4278	bbff
16bits	SRstar(5,2,2,4)	958309036.bdd	421	23,007	1	174c	8a11	ff00
16bits	SRstar(5,2,2,4)	382764202.bdd	838	23,909	1	5555	ab8b	174c
16bits	SRstar(5,2,2,4)	721934437.bdd	2290	25,501	1	94b3	8774	174c
16bits	SRstar(5,2,2,4)	-1446373135.bdd	389	22,958	1	dbc5	ea47	aaaa
16bits	SRstar(5,2,2,4)	-1682520114.bdd	1956	25,47	1	dbc5	1201	bbff
16bits	SRstar(5,2,2,4)	723458011.bdd	428	22,998	1	5555	ac36	5555
16bits	SRstar(5,2,2,4)	-1774582168.bdd	373	22,936	1	bbff	6471	aaaa
16bits	SRstar(5,2,2,4)	1172629401.bdd	1536	25,049	1	ff00	a85e	5555
16bits	SRstar(5,2,2,4)	-2041951243.bdd	438	23,109	1	bbff	56f0	ff00
16bits	SRstar(5,2,2,4)	2088986362.bdd	433	23,176	1	aaaa	faf1	5555
16bits	SRstar(5,2,2,4)	-1000378111.bdd	746	24,434	1	00ff	039e	dbc5
16bits	SRstar(5,2,2,4)	1983037794.bdd	428	23,188	1	aaaa	f2f8	aaaa
16bits	SRstar(5,2,2,4)	-910329488.bdd	1141	24,14	1	ff00	1bba	ff00
16bits	SRstar(5,2,2,4)	712609918.bdd	441	23,044	1	bbff	305f	94b3
16bits	SRstar(5,2,2,4)	-1144659100.bdd	438	23,059	1	94b3	07bc	5555
16bits	SRstar(5,2,2,4)	1363830826.bdd	422	23,006	1	174c	81d6	bbff
16bits	SRstar(5,2,2,4)	1174126278.bdd	356	23,06	1	aaaa	82bb	ff00
16bits	SRstar(5,2,2,4)	631096774.bdd	380	23,02	1	5555	bd80	ff00
16bits	SRstar(5,2,2,4)	1578114204.bdd	403	22,742	1	aaaa	4bc9	bbff
16bits	SRstar(5,2,2,4)	1787304584.bdd	1534	25,048	1	ff00	22a9	00ff
16bits	SRstar(5,2,2,4)	1501689895.bdd	1917	25,516	1	aaaa	d470	dbc5
16bits	SRstar(5,2,2,4)	72686829.bdd	399	22,734	1	5555	8b81	dbc5
16bits	SRstar(5,2,2,4)	1328234473.bdd	415	23,006	1	174c	8a69	00ff
16bits	SRstar(5,2,2,4)	-827980012.bdd	755	24,424	1	00ff	aec7	94b3
16bits	SRstar(5,2,2,4)	936474278.bdd	1191	24,14	1	ff00	d042	dbc5
16bits	SRstar(5,2,2,4)	1140667205.bdd	2041	25,514	1	dbc5	ba9e	ff00
16bits	SRstar(5,2,2,4)	1322498176.bdd	1125	24,511	1	ff00	cb12	aaaa
16bits	SRstar(5,2,2,4)	-1634414887.bdd	435	23,045	1	aaaa	a345	94b3
16bits	SRstar(5,2,2,4)	-1002477012.bdd	418	23,002	1	5555	69f7	00ff
16bits	SRstar(5,2,2,4)	1499936664.bdd	431	23,034	1	bbff	1669	bbff
16bits	SRstar(5,2,2,4)	-1597376838.bdd	1345	24,741	1	ff00	0d64	94b3
16bits	SRstar(5,2,2,4)	-509100808.bdd	719	24,418	1	00ff	4ab2	174c
16bits	SRstar(5,2,2,4)	1131231519.bdd	714	24,445	1	00ff	6f1b	00ff
16bits	SRstar(5,2,2,4)	-1327965370.bdd	1872	25,514	1	bbff	b375	174c
16bits	SRstar(5,2,2,4)	-1639596009.bdd	2037	25,518	1	dbc5	d1cd	174c
16bits	SRstar(5,2,2,4)	1738165623.bdd	693	24,181	1	00ff	1527	aaaa
16bits	SRstar(5,2,2,4)	-225658626.bdd	362	22,981	1	5555	3b76	94b3
16bits	SRstar(5,2,2,4)	1492991979.bdd	684	23,735	1	94b3	efff	00ff
16bits	SRstar(5,2,2,4)	-206399321.bdd	899	24,042	1	94b3	8393	dbc5
16bits	SRstar(5,2,2,4)	1965059597.bdd	398	22,793	1	dbc5	783a	94b3
16bits	SRstar(5,2,2,4)	-1091678514.bdd	431	23,064	1	dbc5	e86e	00ff
16bits	SRstar(5,2,2,4)	229332947.bdd	425	23,012	1	5555	984b	aaaa
16bits	SRstar(5,2,2,4)	1641454672.bdd	1910	25,517	1	aaaa	d3cd	00ff
16bits	SRstar(5,2,2,4)	1440383459.bdd	765	24,386	1	00ff	090d	ff00
16bits	SRstar(5,2,2,4)	842516842.bdd	718	24,356	1	00ff	f27a	5555
16bits	SRstar(5,2,2,4)	-1754589561.bdd	371	22,999	1	aaaa	d981	174c
16bits	SRstar(5,2,2,4)	680852382.bdd	438	23,078	1	dbc5	f822	5555
16bits	SRstar(5,2,2,4)	-1305067570.bdd	435	23,082	1	bbff	a4ba	00ff
16bits	SRstar(6,1,4,4)	-1568378994.bdd	512	23,947	1	dbc5	0310	aaaa
16bits	SRstar(6,1,4,4)	-93085445.bdd	598	23,688	1	5555	fb8a	5555
16bits	SRstar(6,1,4,4)	-420386231.bdd	527	24,006	1	174c	c209	94b3
16bits	SRstar(6,1,4,4)	1684229257.bdd	607	23,701	1	5555	18a1	bbff
16bits	SRstar(6,1,4,4)	-1154989306.bdd	509	23,618	1	5555	dcfa	00ff
16bits	SRstar(6,1,4,4)	1830572364.bdd	927	24,514	1	94b3	dc5f	dbc5
16bits	SRstar(6,1,4,4)	-806385425.bdd	862	24,188	1	94b3	3044	ff00
16bits	SRstar(6,1,4,4)	662416538.bdd	205	23,354	1	ff00	7123	174c
16bits	SRstar(6,1,4,4)	-1060982507.bdd	526	23,97	1	dbc5	74ca	00ff
16bits	SRstar(6,1,4,4)	1152627028.bdd	507	23,794	1	174c	d901	174c
16bits	SRstar(6,1,4,4)	1508199183.bdd	554	23,697	1	dbc5	9c14	94b3
16bits	SRstar(6,1,4,4)	2119652254.bdd	1008	24,608	1	94b3	bd9e	aaaa
16bits	SRstar(6,1,4,4)	-740147811.bdd	500	23,798	1	aaaa	2863	ff00
16bits	SRstar(6,1,4,4)	-1028488178.bdd	645	23,724	1	dbc5	5d1d	ff00
16bits	SRstar(6,1,4,4)	-912741365.bdd	163	23,494	1	ff00	e9f3	94b3
16bits	SRstar(6,1,4,4)	-968143343.bdd	623	23,791	1	aaaa	f489	aaaa
16bits	SRstar(6,1,4,4)	933835504.bdd	600	23,889	1	174c	b10f	bbff
16bits	SRstar(6,1,4,4)	-2066762666.bdd	794	24,124	1	94b3	90dc	00ff
16bits	SRstar(6,1,4,4)	-1823199525.bdd	164	23,494	1	ff00	e93d	00ff
16bits	SRstar(6,1,4,4)	939522083.bdd	573	23,939	1	174c	7290	ff00
16bits	SRstar(6,1,4,4)	1860771129.bdd	871	24,247	1	94b3	3651	94b3

Figure B.6: Raw data for the 16 bit systems.

16bits	SRstar(6,1,4,4)	-397910848.bdd	819	24,091	1	00ff	a25d	aaaa
16bits	SRstar(6,1,4,4)	1998020642.bdd	506	23,632	1	5555	e696	dbc5
16bits	SRstar(6,1,4,4)	883062747.bdd	522	23,69	1	5555	aecd	ff00
16bits	SRstar(6,1,4,4)	-828579492.bdd	211	23,974	1	ff00	6fa5	ff00
16bits	SRstar(6,1,4,4)	1481662390.bdd	828	24,182	1	00ff	601a	bbff
16bits	SRstar(6,1,4,4)	515183630.bdd	1118	24,954	1	94b3	e3e5	bbff
16bits	SRstar(6,1,4,4)	1360377655.bdd	525	24,034	1	aaaa	bb97	174c
16bits	SRstar(6,1,4,4)	-1410692294.bdd	600	23,732	1	aaaa	4f66	94b3
16bits	SRstar(6,1,4,4)	-2015914392.bdd	162	23,458	1	ff00	7ab0	bbff
16bits	SRstar(6,1,4,4)	-1847081740.bdd	528	23,738	1	174c	73d9	dbc5
16bits	SRstar(6,1,4,4)	1163353767.bdd	615	23,728	1	dbc5	1c7c	174c
16bits	SRstar(6,1,4,4)	397944514.bdd	630	23,804	1	174c	b610	00ff
16bits	SRstar(6,1,4,4)	1807363588.bdd	208	23,941	1	ff00	6ef3	5555
16bits	SRstar(6,1,4,4)	1125563522.bdd	531	23,677	1	174c	29e1	aaaa
16bits	SRstar(6,1,4,4)	-219534408.bdd	647	23,806	1	5555	f88c	174c
16bits	SRstar(6,1,4,4)	-833678111.bdd	669	23,896	1	5555	f474	94b3
16bits	SRstar(6,1,4,4)	1882700139.bdd	699	23,416	1	94b3	0fee	174c
16bits	SRstar(6,1,4,4)	-1702293580.bdd	459	23,897	1	aaaa	ff62	dbc5
16bits	SRstar(6,1,4,4)	1641196052.bdd	207	23,914	1	ff00	6dc1	dbc5
16bits	SRstar(6,1,4,4)	-1986275585.bdd	505	23,762	1	174c	af2c	5555
16bits	SRstar(6,1,4,4)	947697766.bdd	574	23,769	1	dbc5	4086	dbc5
16bits	SRstar(6,1,4,4)	1991311396.bdd	466	23,817	1	5555	ee40	aaaa
16bits	SRstar(6,1,4,4)	1572385193.bdd	164	23,631	1	ff00	9176	aaaa
16bits	SRstar(6,1,4,4)	-23288081.bdd	522	23,678	1	dbc5	4731	bbff
16bits	SRstar(6,1,4,4)	194797915.bdd	533	23,885	1	aaaa	0544	bbff
16bits	SRstar(6,1,4,4)	-251543248.bdd	692	24,038	1	aaaa	277b	00ff
16bits	SRstar(6,1,4,4)	-1632225332.bdd	639	23,886	1	aaaa	d778	5555
16bits	SRstar(6,2,2,4)	-975990808.bdd	987	24,144	1	94b3	6a9b	dbc5
16bits	SRstar(6,2,2,4)	-1106513277.bdd	935	23,817	1	94b3	7731	174c
16bits	SRstar(6,2,2,4)	-1284882841.bdd	829	24,197	1	174c	32ba	dbc5
16bits	SRstar(6,2,2,4)	446787315.bdd	1049	24,783	1	aaaa	94df	dbc5
16bits	SRstar(6,2,2,4)	1685224536.bdd	1053	24,646	1	aaaa	4837	00ff
16bits	SRstar(6,2,2,4)	-610912093.bdd	906	24,158	1	bbff	ee30	aaaa
16bits	SRstar(6,2,2,4)	895222451.bdd	911	23,591	1	5555	e1bc	ff00
16bits	SRstar(6,2,2,4)	1218684056.bdd	1027	23,565	1	dbc5	1735	dbc5
16bits	SRstar(6,2,2,4)	1827549806.bdd	866	24,091	1	5555	7a08	aaaa
16bits	SRstar(6,2,2,4)	2047730236.bdd	1049	24,712	1	aaaa	343b	94b3
16bits	SRstar(6,2,2,4)	794244790.bdd	891	26,133	0	00ff	b419	dbc5
16bits	SRstar(6,2,2,4)	-1713624690.bdd	1664	24,14	1	ff00	24c3	5555
16bits	SRstar(6,2,2,4)	1584333296.bdd	915	23,746	1	bbff	7850	94b3
16bits	SRstar(6,2,2,4)	-1952184093.bdd	1030	24,275	1	dbc5	1b73	aaaa
16bits	SRstar(6,2,2,4)	1100473349.bdd	974	23,873	1	174c	4cd1	94b3
16bits	SRstar(6,2,2,4)	-2103509677.bdd	940	23,649	1	174c	2f3a	ff00
16bits	SRstar(6,2,2,4)	-368625008.bdd	876	23,386	1	5555	886f	bbff
16bits	SRstar(6,2,2,4)	751959252.bdd	916	24,21	1	dbc5	1f29	5555
16bits	SRstar(6,2,2,4)	2059665119.bdd	1020	24,344	1	94b3	7fec	bbff
16bits	SRstar(6,2,2,4)	-1117521764.bdd	754	24,008	1	bbff	e58a	174c
16bits	SRstar(6,2,2,4)	386775495.bdd	950	23,757	1	bbff	b429	00ff
16bits	SRstar(6,2,2,4)	-1763465808.bdd	1291	24,503	1	bbff	1add	5555
16bits	SRstar(6,2,2,4)	-34580358.bdd	847	24,091	1	174c	8dd3	00ff
16bits	SRstar(6,2,2,4)	1697797590.bdd	997	24,519	1	94b3	49f7	aaaa
16bits	SRstar(6,2,2,4)	-1875551484.bdd	1047	24,291	1	dbc5	9769	174c
16bits	SRstar(6,2,2,4)	-932956518.bdd	890	23,617	1	bbff	49d5	dbc5
16bits	SRstar(6,2,2,4)	1749598496.bdd	1016	23,849	1	aaaa	2e6c	5555
16bits	SRstar(6,2,2,4)	-1847635636.bdd	956	24,228	1	94b3	5db3	ff00
16bits	SRstar(6,2,2,4)	-1857577109.bdd	997	23,838	1	dbc5	bece	ff00
16bits	SRstar(6,2,2,4)	-1534064198.bdd	942	23,764	1	bbff	3416	bbff
16bits	SRstar(6,2,2,4)	-532832004.bdd	915	24,153	1	94b3	8c0a	94b3
16bits	SRstar(6,2,2,4)	-891097305.bdd	822	23,833	1	aaaa	5d32	174c
16bits	SRstar(6,2,2,4)	223766550.bdd	958	23,614	1	94b3	ccc9	5555
16bits	SRstar(6,2,2,4)	-640835648.bdd	814	24,205	1	5555	92b9	dbc5
16bits	SRstar(6,2,2,4)	1523717880.bdd	1004	23,92	1	5555	8d9a	94b3
16bits	SRstar(6,2,2,4)	412580699.bdd	878	23,984	1	94b3	75d7	00ff
16bits	SRstar(6,2,2,4)	837199940.bdd	873	24,053	1	bbff	7971	ff00
16bits	SRstar(6,2,2,4)	-756160026.bdd	915	23,763	1	aaaa	e24a	bbff
16bits	SRstar(6,2,2,4)	-1403124802.bdd	829	24,018	1	aaaa	e5d6	ff00
16bits	SRstar(6,2,2,4)	499635711.bdd	977	24,249	1	174c	91e8	174c
16bits	SRstar(6,2,2,4)	-2028483296.bdd	772	23,995	1	5555	feef	5555
16bits	SRstar(6,2,2,4)	2135033965.bdd	855	24	1	174c	834a	bbff
16bits	SRstar(6,2,2,4)	-918569101.bdd	913	24,134	1	174c	f8d4	5555
16bits	SRstar(6,2,2,4)	952930330.bdd	926	23,549	1	dbc5	79ee	94b3
16bits	SRstar(6,2,2,4)	1083090090.bdd	784	24,022	1	5555	9f86	174c
16bits	SRstar(6,2,2,4)	-311982664.bdd	970	24,245	1	dbc5	c17a	bbff

Figure B.7: Raw data for the 16 bit systems.

16bits	SRstar(6,2,2,4)	-1026466548.bdd	1671	24,178	1	ff00	7106	94b3
16bits	SRstar(6,2,2,4)	1911925969.bdd	1054	24,591	1	dbc5	ef31	00ff
16bits	SRstar(6,2,2,4)	520469481.bdd	895	26,133	0	00ff	5590	aaaa
16bits	SRstar(6,2,2,4)	2014459757.bdd	949	23,807	1	5555	5f13	00ff
16bits	SRstar(6,2,2,4)	-1210909227.bdd	1014	23,88	1	174c	0220	aaaa
16bits	SRstar(6,2,2,4)	-1964442753.bdd	906	23,588	1	aaaa	8878	aaaa
16bits	SRstar(7,1,4,4)	414182372.bdd	112	26,187	0	ff00	5e23	174c
16bits	SRstar(7,1,4,4)	1065420835.bdd	2733	25,277	1	94b3	60c2	5555
16bits	SRstar(7,1,4,4)	-107985201.bdd	111	26,187	0	ff00	49d0	00ff
16bits	SRstar(7,1,4,4)	12459042.bdd	116	26,187	0	ff00	5124	aaaa
16bits	SRstar(7,1,4,4)	-1323083245.bdd	114	26,187	0	ff00	98e3	ff00
16bits	SRstar(7,1,4,4)	392642878.bdd	115	26,187	0	ff00	ce7c	94b3
16bits	SRstar(7,1,4,4)	411290652.bdd	118	26,187	0	ff00	c2af	dbc5
16bits	SRstar(7,1,4,4)	-177874554.bdd	115	26,187	0	ff00	8343	bbff
16bits	SRstar(7,1,4,4)	-279168845.bdd	118	26,187	0	ff00	76f4	5555
16bits	SRstar(7,1,4,4)	301402789.bdd	2725	25,235	1	bbff	fe62	aaaa
16bits	SRstar(7,2,2,4)	-1279204063.bdd	765	26,103	0	94b3	a4ae	94b3
16bits	SRstar(7,2,2,4)	-612192286.bdd	708	26,104	0	ff00	2231	00ff
16bits	SRstar(7,2,2,4)	313637644.bdd	700	26,104	0	ff00	a84f	174c
16bits	SRstar(7,2,2,4)	1253025598.bdd	704	26,104	0	ff00	2dd4	5555
16bits	SRstar(7,2,2,4)	-1711743268.bdd	1466	26,183	0	aaaa	1126	aaaa
16bits	SRstar(7,2,2,4)	497128952.bdd	703	26,104	0	ff00	6e92	aaaa
16bits	SRstar(7,2,2,4)	983332496.bdd	784	26,103	0	174c	b31e	aaaa
16bits	SRstar(7,2,2,4)	-1097855808.bdd	1400	26,183	0	aaaa	2a1b	5555
16bits	SRstar(7,2,2,4)	781467014.bdd	1443	26,183	0	aaaa	7eab	174c
16bits	SRstar(7,2,2,4)	-1899079327.bdd	718	26,104	0	ff00	0a20	dbc5
16bits	SRstar(7,2,2,4)	1851477440.bdd	1451	26,183	0	aaaa	6e67	dbc5
16bits	SRstar(7,2,2,4)	2039269896.bdd	2753	25,422	1	00ff	c5c5	94b3
16bits	SRstar(7,2,2,4)	558707800.bdd	771	26,103	0	174c	3880	bbff
16bits	SRstar(7,2,2,4)	-936760700.bdd	1410	26,183	0	aaaa	3670	bbff
16bits	SRstar(7,2,2,4)	1990710681.bdd	790	26,103	0	94b3	aa3f	00ff
16bits	SRstar(7,2,2,4)	-865755055.bdd	809	26,103	0	174c	2d07	174c
16bits	SRstar(7,2,2,4)	1881367534.bdd	810	26,103	0	94b3	55ec	ff00
16bits	SRstar(7,2,2,4)	1178393109.bdd	728	26,104	0	ff00	c570	ff00
16bits	SRstar(7,2,2,4)	1147184175.bdd	812	26,103	0	174c	eec5	94b3
16bits	SRstar(7,2,2,4)	1161658989.bdd	810	26,103	0	94b3	3981	5555
16bits	SRstar(7,2,2,4)	1250383075.bdd	807	26,103	0	174c	9f38	ff00
16bits	SRstar(7,2,2,4)	2022182971.bdd	814	26,103	0	174c	5b7d	5555
16bits	SRstar(7,2,2,4)	-740398304.bdd	2830	25,415	1	dbc5	e676	00ff
16bits	SRstar(7,2,2,4)	1739328967.bdd	1491	26,183	0	aaaa	2391	00ff
16bits	SRstar(7,2,2,4)	-116725312.bdd	1479	26,183	0	aaaa	b251	94b3
16bits	SRstar(7,2,2,4)	-707616466.bdd	822	26,103	0	94b3	963d	dbc5
16bits	SRstar(7,2,2,4)	1119538840.bdd	733	26,104	0	ff00	7f05	94b3
16bits	SRstar(7,2,2,4)	2045024554.bdd	799	26,103	0	94b3	ef34	bbff
16bits	SRstar(7,2,2,4)	-249999006.bdd	746	26,104	0	ff00	8710	bbff
16bits	SRstar(7,2,2,4)	-1707331575.bdd	810	26,103	0	174c	6bb2	dbc5
16bits	SRstar(7,2,2,4)	1196369899.bdd	1506	26,183	0	aaaa	583e	ff00
16bits	SRstar(7,2,2,4)	-1051930352.bdd	806	26,103	0	174c	df14	00ff
16bits	SRstar(8,1,4,4)	-1780401386.bdd	456	26,001	0	174c	d53a	bbff
16bits	SRstar(8,1,4,4)	-1813068395.bdd	445	26,001	0	174c	a908	aaaa
16bits	SRstar(8,2,2,4)	-9526814.bdd	3319	24,716	1	ff00	d6a9	94b3
16bits	SRstar(8,2,2,4)	1109430121.bdd	3202	24,749	1	ff00	b95e	dbc5
16bits	SRstar(8,2,2,4)	706176863.bdd	3756	24,943	1	ff00	f4e8	5555
16bits	SRstar(8,2,2,4)	-750177334.bdd	3304	24,758	1	ff00	27c4	aaaa
16bits	SRstar(9,1,4,4)	877036381.bdd	785	26,06	0	ff00	cf77	bbff

Figure B.8: Raw data for the 16 bit systems.

Appendix C

Raw Data for 32 Bit Systems

Key size	System	ID	Time (s)	Nodes (2^n)	1 if solved	plaintext	ciphertext	key
32bits	SRstar(3,2,4,4)	866813617.bdd	150	24,312	0	94b3de7f	608cc99e	94b3de7f
32bits	SRstar(3,2,4,4)	308055624.bdd	86	24,157	0	174ca832	61dae550	aaaaaaaa
32bits	SRstar(3,2,4,4)	791388415.bdd	114	24,301	0	174ca832	c33e2837	55555555
32bits	SRstar(3,2,4,4)	-1862325759.bdd	106	24,395	0	dbc5a241	d585c860	bbbbffff
32bits	SRstar(3,2,4,4)	-246928831.bdd	47	24,216	0	55555555	8e0580a6	0000ffff
32bits	SRstar(3,2,4,4)	-1878923816.bdd	28	24,485	0	55555555	f9ef200f	aaaaaaaa
32bits	SRstar(3,2,4,4)	1398789331.bdd	37	24,216	0	55555555	738023a5	94b3de7f
32bits	SRstar(3,2,4,4)	1512992392.bdd	156	24,295	0	0000ffff	12abb811	55555555
32bits	SRstar(3,2,4,4)	1014042610.bdd	99	24,171	0	ffff0000	59bfd61e	ffff0000
32bits	SRstar(3,2,4,4)	-1526228221.bdd	59	24,652	0	bbbbffff	69de90aa	dbc5a241
32bits	SRstar(3,2,4,4)	2124272418.bdd	190	24,434	0	55555555	e78482a7	bbbbffff
32bits	SRstar(3,2,4,4)	2048487260.bdd	109	24,293	0	dbc5a241	b7533a45	ffff0000
32bits	SRstar(3,2,4,4)	-906286794.bdd	145	24,091	0	94b3de7f	c822441b	bbbbffff
32bits	SRstar(3,2,4,4)	-1071215211.bdd	89	24,046	0	174ca832	8ad7b5d6	174ca832
32bits	SRstar(3,2,4,4)	1239107745.bdd	163	23,769	1	bbbbffff	cb378400	bbbbffff
32bits	SRstar(3,2,4,4)	1050666266.bdd	51	23,922	1	aaaaaaaa	7694a4c8	94b3de7f
32bits	SRstar(3,2,4,4)	-437595301.bdd	51	23,922	1	aaaaaaaa	4fd0649b	bbbbffff
32bits	SRstar(3,2,4,4)	1560645760.bdd	101	24,154	0	dbc5a241	dac8b170	94b3de7f
32bits	SRstar(3,2,4,4)	-1557951550.bdd	163	24,321	0	94b3de7f	93963611	55555555
32bits	SRstar(3,2,4,4)	-1790849786.bdd	54	24,59	0	bbbbffff	0b39fa58	94b3de7f
32bits	SRstar(3,2,4,4)	-518099671.bdd	101	24,24	0	aaaaaaaa	e2021d1d	dbc5a241
32bits	SRstar(3,2,4,4)	759115024.bdd	89	24,079	0	dbc5a241	bb0b917f	174ca832
32bits	SRstar(3,2,4,4)	1950054254.bdd	154	23,563	1	94b3de7f	fb37c4fc	aaaaaaaa
32bits	SRstar(3,2,4,4)	-724107493.bdd	85	24,286	0	ffff0000	2097ee05	94b3de7f
32bits	SRstar(3,2,4,4)	764643408.bdd	118	24,26	0	dbc5a241	09d774d9	0000ffff
32bits	SRstar(3,2,4,4)	-1217966007.bdd	121	24,361	0	174ca832	773a1e11	bbbbffff
32bits	SRstar(3,2,4,4)	2107054891.bdd	26	24,536	0	ffff0000	ab6b1647	aaaaaaaa
32bits	SRstar(3,2,4,4)	-1609648258.bdd	95	24,223	0	ffff0000	77f862e7	dbc5a241
32bits	SRstar(3,2,4,4)	-1851340252.bdd	86	24,191	0	ffff0000	9898ca22	55555555
32bits	SRstar(3,2,4,4)	-729336631.bdd	66	24,174	0	0000ffff	537499fb	aaaaaaaa
32bits	SRstar(3,2,4,4)	384797757.bdd	103	24,3	0	dbc5a241	1f97667e	55555555
32bits	SRstar(3,2,4,4)	253294772.bdd	114	24,047	0	174ca832	81706d1d	94b3de7f
32bits	SRstar(3,2,4,4)	1225408099.bdd	97	24,165	0	174ca832	0f12c224	0000ffff
32bits	SRstar(3,2,4,4)	1654195061.bdd	66	24,179	0	0000ffff	d61ecf35	ffff0000
32bits	SRstar(3,2,4,4)	-1303904149.bdd	80	24,151	0	bbbbffff	6006e923	ffff0000
32bits	SRstar(3,2,4,4)	-1442363110.bdd	50	24,216	0	55555555	4f8229c6	55555555
32bits	SRstar(3,2,4,4)	1439198379.bdd	48	24,216	0	55555555	62844485	ffff0000
32bits	SRstar(3,2,4,4)	-165443877.bdd	48	24,216	0	55555555	db068216	dbc5a241
32bits	SRstar(3,2,4,4)	-56624439.bdd	93	24,37	0	ffff0000	7837f6bd	0000ffff

Figure C.1: Raw data for the 32 bit systems.

32bits	SRstar(3,2,4,4)	2116700929.bdd	63	24,069	0	94b3de7f	fa4535da	ffff0000
32bits	SRstar(3,2,4,4)	591115861.bdd	73	24,197	0	0000ffff	a4097188	174ca832
32bits	SRstar(3,2,4,4)	-1580939223.bdd	48	24,487	0	aaaaaaaa	379c5718	55555555
32bits	SRstar(3,2,4,4)	607164889.bdd	97	24,335	0	dbc5a241	30033899	aaaaaaaa
32bits	SRstar(3,2,4,4)	-1428414212.bdd	123	24,042	0	94b3de7f	db5662be	dbc5a241
32bits	SRstar(3,2,4,4)	1028701702.bdd	74	24,123	0	174ca832	205c7284	dbc5a241
32bits	SRstar(3,2,4,4)	2075250781.bdd	65	24,134	0	0000ffff	2068a43c	0000ffff
32bits	SRstar(3,2,4,4)	-272865652.bdd	54	23,527	1	0000ffff	e05a6c9e	94b3de7f
32bits	SRstar(3,2,4,4)	24303426.bdd	171	23,723	1	55555555	cf13b0c9	174ca832
32bits	SRstar(3,2,4,4)	833112377.bdd	80	24,163	0	bbbbffff	30f2bafe	55555555
32bits	SRstar(3,2,4,4)	1594877367.bdd	150	24,213	0	aaaaaaaa	05ce0948	174ca832
32bits	SRstar(3,2,4,4)	1980255000.bdd	122	24,058	0	94b3de7f	5ca263a4	174ca832
32bits	SRstar(3,2,4,4)	2128018016.bdd	317	25,159	0	ffff0000	2df89720	bbbbffff
32bits	SRstar(3,2,4,4)	-112505789.bdd	63	24,156	0	0000ffff	2e46814c	bbbbffff
32bits	SRstar(3,2,4,4)	-269521588.bdd	74	24,167	0	0000ffff	04541aed	dbc5a241
32bits	SRstar(3,2,4,4)	1251403808.bdd	77	24,148	0	bbbbffff	99317d1f	0000ffff
32bits	SRstar(3,2,4,4)	602480804.bdd	142	24,48	0	dbc5a241	04e83224	dbc5a241
32bits	SRstar(3,2,4,4)	1817265285.bdd	122	24,206	0	174ca832	22b3d7bd	ffff0000
32bits	SRstar(3,2,4,4)	-1008527876.bdd	47	23,922	1	aaaaaaaa	346a5a6b	ffff0000
32bits	SRstar(3,2,4,4)	-800233613.bdd	80	24,176	0	bbbbffff	5df3b47a	174ca832
32bits	SRstar(3,2,4,4)	1072752201.bdd	65	24,011	0	bbbbffff	6ddac90f	aaaaaaaa
32bits	SRstar(3,2,4,4)	2093680366.bdd	26	24,301	0	ffff0000	7c893b45	174ca832
32bits	SRstar(3,2,4,4)	-1706759348.bdd	20	24,223	0	aaaaaaaa	3dedba05	aaaaaaaa
32bits	SRstar(3,2,4,4)	2136101030.bdd	48	23,922	1	aaaaaaaa	0249ff3f	0000ffff
32bits	SRstar(3,2,4,4)	1686808606.bdd	139	23,563	1	94b3de7f	db6854dc	0000ffff
32bits	SRstar(3,4,2,4)	-203286336.bdd	83	24,972	0	dbc5a241	dd5be897	0000ffff
32bits	SRstar(3,4,2,4)	-1865574319.bdd	32	24,985	0	dbc5a241	af2faaba	174ca832
32bits	SRstar(3,4,2,4)	1068305757.bdd	59	24,976	0	94b3de7f	d431e673	94b3de7f
32bits	SRstar(3,4,2,4)	109374084.bdd	36	24,994	0	55555555	e2bffff1	ffff0000
32bits	SRstar(3,4,2,4)	241073700.bdd	37	24,994	0	55555555	d5fe3fbf	174ca832
32bits	SRstar(3,4,2,4)	891552290.bdd	34	24,995	0	bbbbffff	c88ffb87	55555555
32bits	SRstar(3,4,2,4)	1103889821.bdd	34	24,995	0	bbbbffff	ae7e007d	94b3de7f
32bits	SRstar(3,4,2,4)	-1403299813.bdd	139	24,003	0	0000ffff	81bccce0	aaaaaaaa
32bits	SRstar(3,4,2,4)	213352704.bdd	20	24,991	0	ffff0000	27159e9d	aaaaaaaa
32bits	SRstar(3,4,2,4)	-525461326.bdd	91	24,015	0	0000ffff	4248733a	dbc5a241
32bits	SRstar(3,4,2,4)	939910223.bdd	58	24,976	0	94b3de7f	4be54efd	ffff0000
32bits	SRstar(3,4,2,4)	1776837815.bdd	36	24,994	0	174ca832	44e876ca	94b3de7f
32bits	SRstar(3,4,2,4)	-1423879474.bdd	94	24,609	0	aaaaaaaa	ff0d4ff2	aaaaaaaa
32bits	SRstar(3,4,2,4)	497828842.bdd	52	24,976	0	55555555	a38b9652	55555555
32bits	SRstar(3,4,2,4)	88369412.bdd	34	24,995	0	bbbbffff	60650c1a	aaaaaaaa
32bits	SRstar(3,4,2,4)	-1855795406.bdd	36	24,995	0	55555555	3c127642	0000ffff
32bits	SRstar(3,4,2,4)	-336224631.bdd	100	24,484	0	ffff0000	486c0c0c	0000ffff
32bits	SRstar(3,4,2,4)	-1136301097.bdd	34	24,995	0	bbbbffff	43c54f39	ffff0000
32bits	SRstar(3,4,2,4)	301136263.bdd	37	24,995	0	dbc5a241	cfb622dd	ffff0000
32bits	SRstar(3,4,2,4)	1481583501.bdd	112	24,551	0	174ca832	e269941d	55555555
32bits	SRstar(3,4,2,4)	1666229741.bdd	37	24,994	0	174ca832	7d393916	bbbbffff
32bits	SRstar(3,4,2,4)	94858912.bdd	22	24,01	0	0000ffff	7f504c29	bbbbffff
32bits	SRstar(3,4,2,4)	-503218866.bdd	38	24,994	0	55555555	b5ff2052	94b3de7f
32bits	SRstar(3,4,2,4)	2014481207.bdd	180	24,004	0	bbbbffff	cf42bc1c	174ca832
32bits	SRstar(3,4,2,4)	1332792447.bdd	34	24,995	0	bbbbffff	39049495	0000ffff
32bits	SRstar(3,4,2,4)	-38713751.bdd	37	24,995	0	94b3de7f	bc7429b0	0000ffff
32bits	SRstar(3,4,2,4)	-1625693982.bdd	51	24,976	0	55555555	07c70f6c	bbbbffff
32bits	SRstar(3,4,2,4)	1122335407.bdd	80	24,012	0	bbbbffff	e5869fd8	dbc5a241
32bits	SRstar(3,4,2,4)	640851532.bdd	36	24,994	0	174ca832	f6ae239c	0000ffff
32bits	SRstar(3,4,2,4)	-240434201.bdd	65	24,024	0	0000ffff	1c74e054	55555555
32bits	SRstar(3,4,2,4)	-1475304185.bdd	37	24,995	0	aaaaaaaa	ade62434	dbc5a241
32bits	SRstar(3,4,2,4)	-317528284.bdd	37	24,995	0	dbc5a241	73d6a393	aaaaaaaa
32bits	SRstar(3,4,2,4)	687875212.bdd	37	24,995	0	dbc5a241	7dfe1643	dbc5a241
32bits	SRstar(3,4,2,4)	2082378538.bdd	37	24,996	0	55555555	8608de7f	aaaaaaaa
32bits	SRstar(3,4,2,4)	1532611822.bdd	20	24,991	0	ffff0000	a83ff185	55555555
32bits	SRstar(3,4,2,4)	2006681337.bdd	110	24,06	0	0000ffff	76ed1468	94b3de7f
32bits	SRstar(3,4,2,4)	908539847.bdd	37	24,995	0	94b3de7f	da308146	bbbbffff
32bits	SRstar(3,4,2,4)	-964671630.bdd	94	24,376	0	ffff0000	48c05099	dbc5a241
32bits	SRstar(3,4,2,4)	-1508546878.bdd	37	24,994	0	174ca832	265804ca	aaaaaaaa
32bits	SRstar(3,4,2,4)	-1691110539.bdd	37	24,995	0	aaaaaaaa	904ace27	0000ffff
32bits	SRstar(3,4,2,4)	-1119810339.bdd	84	24,978	0	aaaaaaaa	3ecbcb6e	94b3de7f
32bits	SRstar(3,4,2,4)	831452393.bdd	37	24,995	0	dbc5a241	ab2cee98	94b3de7f
32bits	SRstar(3,4,2,4)	-1283799139.bdd	36	24,995	0	aaaaaaaa	d20d13cb	55555555
32bits	SRstar(3,4,2,4)	455412919.bdd	83	24,978	0	aaaaaaaa	abadef3b	174ca832
32bits	SRstar(3,4,2,4)	-405422360.bdd	84	24,972	0	dbc5a241	c7a97dda	55555555
32bits	SRstar(3,4,2,4)	-1441222427.bdd	98	24,063	0	0000ffff	257f5b51	0000ffff
32bits	SRstar(3,4,2,4)	-1812044970.bdd	20	24,991	0	ffff0000	7e457200	bbbbffff
32bits	SRstar(3,4,2,4)	369677124.bdd	61	24,976	0	94b3de7f	82a7b40f	55555555

Figure C.2: Raw data for the 32 bit systems.

32bits	SRstar(3,4,2,4)	1382216930.bdd	100	24,496	0	ffff0000	3097b0aa	174ca832
32bits	SRstar(3,4,2,4)	-1343289191.bdd	18	24,582	0	aaaaaaaa	00d9de2c	ffff0000
32bits	SRstar(3,4,2,4)	326089274.bdd	14	24,582	0	ffff0000	945070a4	94b3de7f
32bits	SRstar(3,4,2,4)	-1231554500.bdd	36	24,995	0	94b3de7f	f1f95d2c	dbc5a241
32bits	SRstar(3,4,2,4)	121251005.bdd	84	24,083	0	ffff0000	390ae331	ffff0000
32bits	SRstar(3,4,2,4)	-814844992.bdd	61	24,976	0	94b3de7f	71072635	174ca832
32bits	SRstar(3,4,2,4)	-2013997961.bdd	37	24,994	0	174ca832	940708d9	174ca832
32bits	SRstar(3,4,2,4)	1480059095.bdd	149	24,114	0	0000ffff	21386c80	ffff0000
32bits	SRstar(3,4,2,4)	510319216.bdd	37	24,994	0	174ca832	1dde0788	ffff0000
32bits	SRstar(3,4,2,4)	-166293338.bdd	37	24,994	0	174ca832	94e34bf2	dbc5a241
32bits	SRstar(3,4,2,4)	1276050359.bdd	18	24,581	0	bbbbffff	81fb670c	bbbbffff
32bits	SRstar(3,4,2,4)	-1900207751.bdd	53	24,007	0	0000ffff	3ad7d7db	174ca832
32bits	SRstar(3,4,2,4)	-1936746675.bdd	87	24,083	0	55555555	6b951208	dbc5a241
32bits	SRstar(3,4,2,4)	-127812911.bdd	37	24,995	0	dbc5a241	9b18ff85	bbbbffff
32bits	SRstar(3,4,2,4)	136249116.bdd	18	24,582	0	aaaaaaaa	e65f6ea4	bbbbffff
32bits	SRstar(3,4,2,4)	-1270330855.bdd	61	24,976	0	94b3de7f	4673806c	aaaaaaaa
32bits	SRstar(4,2,4,4)	1581540419.bdd	466	24,919	0	174ca832	1d51b9ff	dbc5a241
32bits	SRstar(4,2,4,4)	-1990550675.bdd	370	24,919	0	174ca832	bdb7ec9b	aaaaaaaa
32bits	SRstar(4,2,4,4)	-775185262.bdd	297	24,928	0	dbc5a241	95c0b5e5	0000ffff
32bits	SRstar(4,2,4,4)	646523050.bdd	26	24,711	0	aaaaaaaa	308b641b	55555555
32bits	SRstar(4,2,4,4)	-493571730.bdd	406	25,042	0	dbc5a241	123467e2	aaaaaaaa
32bits	SRstar(4,2,4,4)	-1376522708.bdd	129	24,652	0	0000ffff	303cae25	0000ffff
32bits	SRstar(4,2,4,4)	1419308353.bdd	18	24,099	0	0000ffff	de1fbc04	94b3de7f
32bits	SRstar(4,2,4,4)	464093442.bdd	428	24,916	0	dbc5a241	172a3d45	174ca832
32bits	SRstar(4,2,4,4)	1770942600.bdd	53	24,502	0	ffff0000	6e8e4bc1	55555555
32bits	SRstar(4,2,4,4)	881264768.bdd	25	24,73	0	55555555	26874494	dbc5a241
32bits	SRstar(4,2,4,4)	-760787726.bdd	25	24,719	0	0000ffff	3d6b75a4	aaaaaaaa
32bits	SRstar(4,2,4,4)	564958435.bdd	356	24,919	0	174ca832	524ac15e	55555555
32bits	SRstar(4,2,4,4)	1819150145.bdd	208	24,419	0	0000ffff	e93ff949	55555555
32bits	SRstar(4,2,4,4)	-430833798.bdd	50	24,752	0	174ca832	24e78a21	174ca832
32bits	SRstar(4,2,4,4)	715768870.bdd	47	24,808	0	aaaaaaaa	f5f5b9a1	174ca832
32bits	SRstar(4,2,4,4)	-1461558619.bdd	29	24,717	0	bbbbffff	ad18f687	dbc5a241
32bits	SRstar(4,2,4,4)	675323388.bdd	230	24,434	0	ffff0000	7a09c301	bbbbffff
32bits	SRstar(4,2,4,4)	660894432.bdd	117	24,35	0	bbbbffff	41f7e9ec	aaaaaaaa
32bits	SRstar(4,2,4,4)	-748925253.bdd	415	24,79	0	dbc5a241	270d9bde	55555555
32bits	SRstar(4,2,4,4)	-487521231.bdd	164	24,266	0	bbbbffff	dd5512e3	ffff0000
32bits	SRstar(4,2,4,4)	-1093330000.bdd	163	24,517	0	ffff0000	52dd8963	174ca832
32bits	SRstar(4,2,4,4)	-905487723.bdd	360	24,974	0	94b3de7f	150313db	aaaaaaaa
32bits	SRstar(4,2,4,4)	844009217.bdd	26	24,718	0	0000ffff	9c3ba84a	dbc5a241
32bits	SRstar(4,2,4,4)	-1786035361.bdd	356	24,878	0	94b3de7f	068e50d1	dbc5a241
32bits	SRstar(4,2,4,4)	791367167.bdd	232	24,211	0	94b3de7f	e906a0e9	55555555
32bits	SRstar(4,2,4,4)	1541576852.bdd	334	24,806	0	dbc5a241	4d1d2eb6	ffff0000
32bits	SRstar(4,2,4,4)	406249906.bdd	163	24,454	0	ffff0000	a8639245	0000ffff
32bits	SRstar(4,2,4,4)	-1910641802.bdd	358	24,71	0	dbc5a241	ec326b74	dbc5a241
32bits	SRstar(4,2,4,4)	638013596.bdd	24	24,732	0	bbbbffff	73561246	94b3de7f
32bits	SRstar(4,2,4,4)	1352105105.bdd	204	24,754	0	ffff0000	26db30d2	dbc5a241
32bits	SRstar(4,2,4,4)	1882762524.bdd	38	24,863	0	0000ffff	27c667c9	174ca832
32bits	SRstar(4,2,4,4)	1169818564.bdd	404	24,796	0	dbc5a241	fa841bd2	94b3de7f
32bits	SRstar(4,2,4,4)	-261124066.bdd	95	24,327	0	aaaaaaaa	86e684a8	aaaaaaaa
32bits	SRstar(4,2,4,4)	-2071170808.bdd	51	24,802	0	aaaaaaaa	e67229af	bbbbffff
32bits	SRstar(4,2,4,4)	1652988446.bdd	297	24,919	0	174ca832	c04c34c3	ffff0000
32bits	SRstar(4,2,4,4)	2129737414.bdd	75	24,292	0	aaaaaaaa	8a534c44	94b3de7f
32bits	SRstar(4,2,4,4)	-1905228298.bdd	31	24,733	0	55555555	52f131b1	174ca832
32bits	SRstar(4,2,4,4)	594121106.bdd	17	24,045	0	bbbbffff	344fcfa9	0000ffff
32bits	SRstar(4,2,4,4)	-64526110.bdd	291	24,919	0	174ca832	8b2deb4d	bbbbffff
32bits	SRstar(4,2,4,4)	-244176485.bdd	194	24,674	0	94b3de7f	d0b14314	94b3de7f
32bits	SRstar(4,2,4,4)	-314698335.bdd	199	24,666	0	94b3de7f	7a96e636	bbbbffff
32bits	SRstar(4,2,4,4)	115136379.bdd	26	24,723	0	0000ffff	ec6e17c5	ffff0000
32bits	SRstar(4,2,4,4)	1607374970.bdd	27	24,731	0	55555555	91fdb0ac	ffff0000
32bits	SRstar(4,2,4,4)	71652751.bdd	28	24,725	0	55555555	db2daa85	94b3de7f
32bits	SRstar(4,2,4,4)	-573096932.bdd	220	24,23	0	aaaaaaaa	a54bba12	dbc5a241
32bits	SRstar(4,2,4,4)	-884583603.bdd	232	24,606	0	ffff0000	60f2bcf2	ffff0000
32bits	SRstar(4,2,4,4)	1813212937.bdd	292	24,877	0	94b3de7f	8522570d	ffff0000
32bits	SRstar(4,2,4,4)	-1545759252.bdd	199	24,686	0	bbbbffff	8d669cec	55555555
32bits	SRstar(4,2,4,4)	-492659372.bdd	172	24,262	0	bbbbffff	7496eb11	174ca832
32bits	SRstar(4,2,4,4)	365437325.bdd	30	24,733	0	aaaaaaaa	e300509c	0000ffff
32bits	SRstar(4,2,4,4)	1829548612.bdd	135	24,054	0	94b3de7f	d68b9bdb	174ca832
32bits	SRstar(4,2,4,4)	2109311170.bdd	130	24,417	0	ffff0000	6a0be4e3	94b3de7f
32bits	SRstar(4,2,4,4)	-1558090944.bdd	434	24,919	0	174ca832	39aec4f6	0000ffff
32bits	SRstar(4,2,4,4)	-465330006.bdd	326	24,919	0	174ca832	65369598	94b3de7f
32bits	SRstar(4,2,4,4)	-64962141.bdd	214	24,61	0	ffff0000	fcbb44dd	aaaaaaaa
32bits	SRstar(4,2,4,4)	-62173803.bdd	23	24,102	0	aaaaaaaa	e1afc2f1	ffff0000
32bits	SRstar(4,2,4,4)	21757766.bdd	50	24,752	0	55555555	d903b67e	0000ffff

Figure C.3: Raw data for the 32 bit systems.

32bits	SRstar(4,2,4,4)	-878904165.bdd	262	24,452	0	94b3de7f	01ccaabb	0000ffff
32bits	SRstar(4,2,4,4)	-2127650231.bdd	197	24,599	0	55555555	49fca8ec	aaaaaaaa
32bits	SRstar(4,2,4,4)	-555975757.bdd	167	24,307	0	55555555	9303ded3	bbbbffff
32bits	SRstar(4,2,4,4)	-1813877201.bdd	25	24,728	0	bbbbffff	b25358ee	bbbbffff
32bits	SRstar(4,2,4,4)	891091462.bdd	369	24,85	0	dbc5a241	23dbc936	bbbbffff
32bits	SRstar(4,2,4,4)	1750259366.bdd	37	24,715	0	55555555	5e450009	55555555
32bits	SRstar(4,2,4,4)	-960558418.bdd	25	24,723	0	0000ffff	5216897f	bbbbffff
32bits	SRstar(4,4,2,4)	1887008287.bdd	104	24,905	0	174ca832	4d888be7	0000ffff
32bits	SRstar(4,4,2,4)	1291224545.bdd	57	24,24	0	94b3de7f	c4e67c7d	bbbbffff
32bits	SRstar(4,4,2,4)	1751944579.bdd	88	24,064	0	94b3de7f	024a044e	174ca832
32bits	SRstar(4,4,2,4)	1579229110.bdd	207	24,726	0	174ca832	1f964733	ffff0000
32bits	SRstar(4,4,2,4)	-1015826526.bdd	49	24,208	0	ffff0000	3c86cc7e	dbc5a241
32bits	SRstar(4,4,2,4)	977176564.bdd	197	24,472	0	bbbbffff	dae0a4e6	aaaaaaaa
32bits	SRstar(4,4,2,4)	-1499084401.bdd	50	24,133	0	aaaaaaaa	8773242f	0000ffff
32bits	SRstar(4,4,2,4)	1569631347.bdd	53	24,168	0	aaaaaaaa	07461467	55555555
32bits	SRstar(4,4,2,4)	846901003.bdd	180	24,724	0	bbbbffff	a0996879	dbc5a241
32bits	SRstar(4,4,2,4)	1991572185.bdd	52	24,092	0	174ca832	f1a5a486	dbc5a241
32bits	SRstar(4,4,2,4)	221930811.bdd	65	24,037	0	94b3de7f	b71619ba	aaaaaaaa
32bits	SRstar(4,4,2,4)	436233204.bdd	216	24,882	0	174ca832	5cfe1715	aaaaaaaa
32bits	SRstar(4,4,2,4)	-1656289149.bdd	58	24,747	0	dbc5a241	27f4d9d7	aaaaaaaa
32bits	SRstar(4,4,2,4)	784780534.bdd	228	24,754	0	bbbbffff	32d80aa2	174ca832
32bits	SRstar(4,4,2,4)	-1211501823.bdd	293	24,12	0	0000ffff	542eb4a8	94b3de7f
32bits	SRstar(4,4,2,4)	-1184236704.bdd	119	24,641	0	dbc5a241	cccc1472	ffff0000
32bits	SRstar(4,4,2,4)	-1775045540.bdd	93	24,676	0	55555555	a1cad29a	0000ffff
32bits	SRstar(4,4,2,4)	28736014.bdd	305	24,436	0	0000ffff	269e646e	174ca832
32bits	SRstar(4,4,2,4)	601561999.bdd	26	24,002	0	aaaaaaaa	6d05ab50	dbc5a241
32bits	SRstar(4,4,2,4)	-352938948.bdd	58	24,251	0	bbbbffff	c33e6dd8	bbbbffff
32bits	SRstar(4,4,2,4)	1387382098.bdd	107	24,902	0	94b3de7f	0cc7ae98	55555555
32bits	SRstar(4,4,2,4)	1688100102.bdd	214	24,784	0	174ca832	c8c715a5	94b3de7f
32bits	SRstar(4,4,2,4)	1782131944.bdd	190	24,686	0	94b3de7f	abf4cd04	ffff0000
32bits	SRstar(4,4,2,4)	1665433398.bdd	91	24,58	0	55555555	d94ff422	174ca832
32bits	SRstar(4,4,2,4)	-998143578.bdd	290	24,359	0	0000ffff	de997ae4	ffff0000
32bits	SRstar(4,4,2,4)	1407073924.bdd	101	24,807	0	ffff0000	b609bd74	0000ffff
32bits	SRstar(4,4,2,4)	1394828027.bdd	57	24,35	0	aaaaaaaa	b7e91f07	ffff0000
32bits	SRstar(4,4,2,4)	9420818.bdd	54	24,215	0	ffff0000	839f0b87	174ca832
32bits	SRstar(4,4,2,4)	-1652544068.bdd	59	24,306	0	dbc5a241	7aa6624f	174ca832
32bits	SRstar(4,4,2,4)	-1356430867.bdd	56	24,154	0	174ca832	284e297b	174ca832
32bits	SRstar(4,4,2,4)	813629779.bdd	83	24,451	0	aaaaaaaa	298ffc17	174ca832
32bits	SRstar(4,4,2,4)	249940372.bdd	52	24,091	0	dbc5a241	a4811d29	94b3de7f
32bits	SRstar(4,4,2,4)	1212992798.bdd	57	24,217	0	55555555	dbe5a742	55555555
32bits	SRstar(4,4,2,4)	743085714.bdd	282	24,913	0	aaaaaaaa	38cc1cb9	bbbbffff
32bits	SRstar(4,4,2,4)	-1540986558.bdd	308	24,911	0	aaaaaaaa	323a8eae	aaaaaaaa
32bits	SRstar(4,4,2,4)	1068733499.bdd	65	24,396	0	94b3de7f	ebcbd74d	dbc5a241
32bits	SRstar(4,4,2,4)	-2062224399.bdd	254	24,837	0	dbc5a241	70b67fd5	bbbbffff
32bits	SRstar(4,4,2,4)	-404785526.bdd	63	24,36	0	174ca832	3d5e03eb	bbbbffff
32bits	SRstar(4,4,2,4)	2091591218.bdd	102	24,259	0	bbbbffff	c199c31a	55555555
32bits	SRstar(4,4,2,4)	-1692286675.bdd	60	24,136	0	94b3de7f	5b225102	94b3de7f
32bits	SRstar(4,4,2,4)	-420046598.bdd	82	24,452	0	ffff0000	b5442a04	aaaaaaaa
32bits	SRstar(4,4,2,4)	-246572368.bdd	57	24,213	0	ffff0000	6cae2a3c	55555555
32bits	SRstar(4,4,2,4)	-846268200.bdd	276	24,249	0	0000ffff	c237e525	dbc5a241
32bits	SRstar(4,4,2,4)	-343736215.bdd	188	24,396	0	55555555	3c12bc14	dbc5a241
32bits	SRstar(4,4,2,4)	149857679.bdd	42	24,122	0	55555555	fc0e4652	bbbbffff
32bits	SRstar(4,4,2,4)	236520574.bdd	232	24,712	0	55555555	bdf67a17	aaaaaaaa
32bits	SRstar(4,4,2,4)	955174538.bdd	264	24,915	0	bbbbffff	a3b76a44	94b3de7f
32bits	SRstar(4,4,2,4)	-964072975.bdd	191	24,567	0	dbc5a241	1a5087bc	dbc5a241
32bits	SRstar(4,4,2,4)	538387073.bdd	210	24,725	0	bbbbffff	3a63a973	0000ffff
32bits	SRstar(4,4,2,4)	1213359025.bdd	48	24,093	0	ffff0000	934fcfdd	ffff0000
32bits	SRstar(4,4,2,4)	-1923791425.bdd	309	24,897	0	aaaaaaaa	f29f9afb	94b3de7f
32bits	SRstar(4,4,2,4)	-557407424.bdd	163	24,352	0	ffff0000	c823910b	bbbbffff
32bits	SRstar(4,4,2,4)	848718487.bdd	65	24,354	0	dbc5a241	7f2599c8	0000ffff
32bits	SRstar(4,4,2,4)	-175568777.bdd	64	24,356	0	174ca832	2aa5760b	55555555
32bits	SRstar(4,4,2,4)	1988852410.bdd	119	24,178	0	55555555	fd0c500f	94b3de7f
32bits	SRstar(4,4,2,4)	165242284.bdd	288	24,249	0	0000ffff	7ea776e6	0000ffff
32bits	SRstar(4,4,2,4)	1854195829.bdd	95	24,392	0	94b3de7f	f74fe9a1	0000ffff
32bits	SRstar(4,4,2,4)	-2138570750.bdd	217	24,635	0	ffff0000	45d727ab	94b3de7f
32bits	SRstar(4,4,2,4)	-1756081852.bdd	280	24,919	0	dbc5a241	0a652815	55555555
32bits	SRstar(4,4,2,4)	-528531936.bdd	293	24,959	0	bbbbffff	9832e9e3	ffff0000
32bits	SRstar(4,4,2,4)	822950006.bdd	213	24,722	0	55555555	891e5d39	ffff0000
32bits	SRstar(5,2,4,4)	-561393562.bdd	107	24,786	0	0000ffff	a4105614	94b3de7f
32bits	SRstar(5,2,4,4)	1036138346.bdd	167	24,805	0	dbc5a241	94c4c7e9	bbbbffff
32bits	SRstar(5,2,4,4)	-488573117.bdd	209	24,628	0	aaaaaaaa	881538ed	174ca832
32bits	SRstar(5,2,4,4)	-1554611227.bdd	140	24,708	0	0000ffff	0ae07589	ffff0000
32bits	SRstar(5,2,4,4)	477046689.bdd	133	24,213	0	ffff0000	3b81bd22	174ca832

Figure C.4: Raw data for the 32 bit systems.

32bits	SRstar(5,2,4,4)	311995734.bdd	196	24,948	0	bbbbffff	3ed57817	aaaaaaaa
32bits	SRstar(5,2,4,4)	240575000.bdd	266	24,87	0	dbc5a241	ae499743	ffff0000
32bits	SRstar(5,2,4,4)	1789049522.bdd	66	24,579	0	bbbbffff	7de2ee2f	bbbbffff
32bits	SRstar(5,2,4,4)	-855504814.bdd	439	24,357	0	55555555	88ee479d	ffff0000
32bits	SRstar(5,2,4,4)	-1829892226.bdd	213	24,708	0	0000ffff	761b6e4b	174ca832
32bits	SRstar(5,2,4,4)	-57058683.bdd	88	24,631	0	0000ffff	e8519dcb	dbc5a241
32bits	SRstar(5,2,4,4)	-31411888.bdd	188	24,684	0	aaaaaaaa	a08fa3ab	bbbbffff
32bits	SRstar(5,2,4,4)	-2016678190.bdd	214	24,991	0	bbbbffff	269d8ba6	0000ffff
32bits	SRstar(5,2,4,4)	1155185580.bdd	270	24,692	0	174ca832	c9e7b9aa	aaaaaaaa
32bits	SRstar(5,2,4,4)	-207742857.bdd	101	24,659	0	94b3de7f	4c7039a8	0000ffff
32bits	SRstar(5,2,4,4)	2062182919.bdd	183	24,929	0	aaaaaaaa	957a4e42	94b3de7f
32bits	SRstar(5,2,4,4)	1777489240.bdd	108	24,716	0	0000ffff	43e010cb	55555555
32bits	SRstar(5,2,4,4)	1231966125.bdd	297	24,773	0	aaaaaaaa	a72bbef0	55555555
32bits	SRstar(5,2,4,4)	368084363.bdd	108	24,745	0	bbbbffff	b8cc8b44	174ca832
32bits	SRstar(5,2,4,4)	338592818.bdd	134	24,568	0	174ca832	3b048806	55555555
32bits	SRstar(5,2,4,4)	-1576027243.bdd	121	24,693	0	ffff0000	fdafc0fa	bbbbffff
32bits	SRstar(5,2,4,4)	1900270776.bdd	299	24,943	0	dbc5a241	ea90d414	55555555
32bits	SRstar(5,2,4,4)	331370194.bdd	214	24,44	0	94b3de7f	b011a9f5	174ca832
32bits	SRstar(5,2,4,4)	-1408921692.bdd	448	24,422	0	aaaaaaaa	c60659dc	0000ffff
32bits	SRstar(5,2,4,4)	-1086584034.bdd	101	24,631	0	0000ffff	4c84c406	aaaaaaaa
32bits	SRstar(5,2,4,4)	1310435280.bdd	285	24,66	0	55555555	0f694e06	0000ffff
32bits	SRstar(5,2,4,4)	-310868525.bdd	228	24,367	0	174ca832	28fd19f0	dbc5a241
32bits	SRstar(5,2,4,4)	-640865356.bdd	199	24,525	0	dbc5a241	5b4a258a	94b3de7f
32bits	SRstar(5,2,4,4)	-795980228.bdd	228	24,574	0	aaaaaaaa	debc4136	dbc5a241
32bits	SRstar(5,2,4,4)	-1184274048.bdd	257	24,11	0	174ca832	3ec84cd5	94b3de7f
32bits	SRstar(5,2,4,4)	1812697937.bdd	231	24,527	0	55555555	f3ab0199	bbbbffff
32bits	SRstar(5,2,4,4)	-321953722.bdd	128	24,264	0	94b3de7f	39613dbb	94b3de7f
32bits	SRstar(5,2,4,4)	-1209028421.bdd	245	24,688	0	94b3de7f	33e7431a	55555555
32bits	SRstar(5,2,4,4)	-1329719197.bdd	417	24,913	0	174ca832	78271cd4	0000ffff
32bits	SRstar(5,2,4,4)	1989737961.bdd	188	24,351	0	55555555	7a8010b4	94b3de7f
32bits	SRstar(5,2,4,4)	-1971520113.bdd	122	24,693	0	ffff0000	ba252b54	55555555
32bits	SRstar(5,2,4,4)	-504518097.bdd	149	24,476	0	0000ffff	206b2108	0000ffff
32bits	SRstar(5,2,4,4)	-1381477916.bdd	298	24,862	0	174ca832	3254e680	ffff0000
32bits	SRstar(5,2,4,4)	-1004398626.bdd	122	24,27	0	0000ffff	e92fb859	bbbbffff
32bits	SRstar(5,2,4,4)	1010787570.bdd	117	24,53	0	55555555	62f47bb9	dbc5a241
32bits	SRstar(5,2,4,4)	178739492.bdd	120	24,468	0	bbbbffff	979d95c8	dbc5a241
32bits	SRstar(5,2,4,4)	1107899719.bdd	119	24,693	0	ffff0000	8ee1cc77	ffff0000
32bits	SRstar(5,2,4,4)	260518758.bdd	203	24,909	0	bbbbffff	f3d13bc4	55555555
32bits	SRstar(5,2,4,4)	-1870286688.bdd	247	24,976	0	55555555	3b80ce68	174ca832
32bits	SRstar(5,2,4,4)	-1428541877.bdd	200	24,765	0	174ca832	38a8be89	bbbbffff
32bits	SRstar(5,2,4,4)	995631859.bdd	123	24,693	0	ffff0000	c3b66adb	aaaaaaaa
32bits	SRstar(5,2,4,4)	626247888.bdd	122	24,172	0	94b3de7f	afb8189c	aaaaaaaa
32bits	SRstar(5,2,4,4)	1427356057.bdd	257	24,913	0	55555555	70e23ee4	55555555
32bits	SRstar(5,2,4,4)	346648328.bdd	117	24,152	0	bbbbffff	93227b3f	ffff0000
32bits	SRstar(5,2,4,4)	1197727417.bdd	102	24,592	0	94b3de7f	9dae3a39	bbbbffff
32bits	SRstar(5,2,4,4)	444235113.bdd	198	24,965	0	bbbbffff	56f02747	94b3de7f
32bits	SRstar(5,2,4,4)	715151884.bdd	406	24,862	0	174ca832	2eb72ab5	174ca832
32bits	SRstar(5,2,4,4)	674849087.bdd	252	24,721	0	55555555	3a520c47	aaaaaaaa
32bits	SRstar(5,2,4,4)	-379759231.bdd	186	24,929	0	aaaaaaaa	91e8f208	aaaaaaaa
32bits	SRstar(5,2,4,4)	526579796.bdd	216	24,892	0	dbc5a241	7ef0baf7	dbc5a241
32bits	SRstar(5,2,4,4)	-1151279281.bdd	119	24,693	0	ffff0000	bfbe8b95	94b3de7f
32bits	SRstar(5,2,4,4)	-262006196.bdd	183	24,929	0	aaaaaaaa	fe1208d0	ffff0000
32bits	SRstar(5,2,4,4)	-709532365.bdd	90	24,592	0	94b3de7f	1d6338f8	dbc5a241
32bits	SRstar(5,2,4,4)	2107489994.bdd	201	24,525	0	dbc5a241	779d12b5	174ca832
32bits	SRstar(5,2,4,4)	-1423435862.bdd	236	24,264	0	94b3de7f	ce836d29	ffff0000
32bits	SRstar(5,2,4,4)	1726630223.bdd	261	24,612	0	dbc5a241	45e2fedf	aaaaaaaa
32bits	SRstar(5,2,4,4)	-1461974230.bdd	123	24,693	0	ffff0000	07bc3c64	0000ffff
32bits	SRstar(5,2,4,4)	1089769645.bdd	215	24,819	0	dbc5a241	487f9975	0000ffff
32bits	SRstar(5,2,4,4)	-2024849834.bdd	121	24,693	0	ffff0000	62ffb460	dbc5a241
32bits	SRstar(5,4,2,4)	1732782020.bdd	192	24,538	0	ffff0000	1a04bf37	ffff0000
32bits	SRstar(5,4,2,4)	-1084820820.bdd	173	24,272	0	0000ffff	1fd0cf8f	dbc5a241
32bits	SRstar(5,4,2,4)	1586550753.bdd	92	24,436	0	94b3de7f	6604dea7	aaaaaaaa
32bits	SRstar(5,4,2,4)	1786645622.bdd	147	24,761	0	dbc5a241	6c8cb2bb	0000ffff
32bits	SRstar(5,4,2,4)	1197608450.bdd	96	24,177	0	174ca832	6133ce5a	0000ffff
32bits	SRstar(5,4,2,4)	1448083459.bdd	157	24,526	0	aaaaaaaa	b33fba82	ffff0000
32bits	SRstar(5,4,2,4)	-675804931.bdd	121	24,501	0	174ca832	fe957082	ffff0000
32bits	SRstar(5,4,2,4)	868992721.bdd	214	24,201	0	0000ffff	331b02ea	0000ffff
32bits	SRstar(5,4,2,4)	1186603293.bdd	51	24,076	0	dbc5a241	b3fdb970	dbc5a241
32bits	SRstar(5,4,2,4)	1017343476.bdd	153	24,742	0	ffff0000	407b4513	dbc5a241
32bits	SRstar(5,4,2,4)	1337039168.bdd	55	24,075	0	55555555	4894ba79	bbbbffff
32bits	SRstar(5,4,2,4)	1998075059.bdd	145	24,538	0	ffff0000	fd8795b2	94b3de7f
32bits	SRstar(5,4,2,4)	-1371312261.bdd	166	24,574	0	174ca832	7500e454	94b3de7f
32bits	SRstar(5,4,2,4)	1896416303.bdd	123	24,574	0	94b3de7f	a555240e	dbc5a241

Figure C.5: Raw data for the 32 bit systems.

32bits	SRstar(5,4,2,4)	13341228.bdd	56	24,065	0	dbc5a241	cdcc8ffe	94b3de7f
32bits	SRstar(5,4,2,4)	-155868276.bdd	125	24,859	0	174ca832	685785ec	bbbbffff
32bits	SRstar(5,4,2,4)	387037992.bdd	164	24,582	0	94b3de7f	286e6bc6	bbbbffff
32bits	SRstar(5,4,2,4)	982225469.bdd	90	24,445	0	bbbbffff	30f15a2a	0000ffff
32bits	SRstar(5,4,2,4)	-885923259.bdd	176	24,604	0	aaaaaaaa	4bfa3f54	bbbbffff
32bits	SRstar(5,4,2,4)	2004640728.bdd	169	24,582	0	55555555	1752cedc	94b3de7f
32bits	SRstar(5,4,2,4)	273752045.bdd	51	24,073	0	55555555	3bbbba25	dbc5a241
32bits	SRstar(5,4,2,4)	-113918680.bdd	132	24,538	0	ffff0000	1e531667	bbbbffff
32bits	SRstar(5,4,2,4)	680711377.bdd	163	24,604	0	aaaaaaaa	ad2f4106	174ca832
32bits	SRstar(5,4,2,4)	-1808491624.bdd	137	24,836	0	dbc5a241	7cb3aad8	174ca832
32bits	SRstar(5,4,2,4)	628034746.bdd	57	24,075	0	174ca832	567e3346	55555555
32bits	SRstar(5,4,2,4)	-1135028320.bdd	170	24,569	0	55555555	67effc01	0000ffff
32bits	SRstar(5,4,2,4)	-1205996923.bdd	168	24,185	0	0000ffff	89a5818e	ffff0000
32bits	SRstar(5,4,2,4)	526750509.bdd	168	24,468	0	dbc5a241	af86b297	55555555
32bits	SRstar(5,4,2,4)	140593787.bdd	112	24,467	0	aaaaaaaa	b1c43c1f	aaaaaaaa
32bits	SRstar(5,4,2,4)	1542388943.bdd	310	24,61	0	0000ffff	0de16f89	55555555
32bits	SRstar(5,4,2,4)	1965358686.bdd	149	24,835	0	55555555	ca43d6e8	ffff0000
32bits	SRstar(5,4,2,4)	1971627925.bdd	165	24,467	0	55555555	aa68b71e	55555555
32bits	SRstar(5,4,2,4)	642757625.bdd	165	24,189	0	0000ffff	593e58b0	aaaaaaaa
32bits	SRstar(5,4,2,4)	-795356101.bdd	133	24,538	0	ffff0000	deef33cf	55555555
32bits	SRstar(5,4,2,4)	-505530140.bdd	139	24,538	0	ffff0000	27e7eef6	174ca832
32bits	SRstar(5,4,2,4)	912040708.bdd	62	24,674	0	aaaaaaaa	4771cc21	55555555
32bits	SRstar(5,4,2,4)	386422761.bdd	170	24,604	0	dbc5a241	dc82f2f6	aaaaaaaa
32bits	SRstar(5,4,2,4)	-94020767.bdd	94	24,436	0	94b3de7f	6a61594a	94b3de7f
32bits	SRstar(5,4,2,4)	-849443725.bdd	206	24,215	0	0000ffff	b6a2ce1b	174ca832
32bits	SRstar(5,4,2,4)	760461723.bdd	166	24,522	0	bbbbffff	140c1e72	ffff0000
32bits	SRstar(5,4,2,4)	-1567515681.bdd	55	24,074	0	174ca832	c5d80765	aaaaaaaa
32bits	SRstar(5,4,2,4)	-820073816.bdd	75	24,836	0	bbbbffff	3ec33c28	55555555
32bits	SRstar(5,4,2,4)	1830327507.bdd	141	24,538	0	ffff0000	1d8e049b	aaaaaaaa
32bits	SRstar(5,4,2,4)	-909449088.bdd	54	24,065	0	bbbbffff	2bb12133	dbc5a241
32bits	SRstar(5,4,2,4)	1539288099.bdd	119	24,674	0	94b3de7f	8aebe43b	174ca832
32bits	SRstar(5,4,2,4)	-696240372.bdd	56	24,071	0	bbbbffff	99a5d131	174ca832
32bits	SRstar(5,4,2,4)	1528851441.bdd	177	24,604	0	bbbbffff	f2db8b83	aaaaaaaa
32bits	SRstar(5,4,2,4)	1548137842.bdd	168	24,574	0	aaaaaaaa	5592cca6	0000ffff
32bits	SRstar(5,4,2,4)	-1995407539.bdd	172	24,604	0	55555555	7bdd835b	aaaaaaaa
32bits	SRstar(5,4,2,4)	1535741499.bdd	89	24,436	0	94b3de7f	60e9bb2e	0000ffff
32bits	SRstar(5,4,2,4)	202350845.bdd	76	24,262	0	174ca832	4e685262	dbc5a241
32bits	SRstar(5,4,2,4)	-1669659307.bdd	68	24,668	0	dbc5a241	6aae8e8e	bbbbffff
32bits	SRstar(5,4,2,4)	-920534754.bdd	175	24,586	0	aaaaaaaa	6de237bc	dbc5a241
32bits	SRstar(5,4,2,4)	-1112619872.bdd	57	24,071	0	bbbbffff	3c6e79ba	bbbbffff
32bits	SRstar(5,4,2,4)	-1158835414.bdd	53	24,073	0	94b3de7f	0122c6ec	55555555
32bits	SRstar(5,4,2,4)	-1455759848.bdd	133	24,802	0	94b3de7f	df543fa1	ffff0000
32bits	SRstar(5,4,2,4)	981091692.bdd	211	24,256	0	0000ffff	3bcf7027	94b3de7f
32bits	SRstar(5,4,2,4)	227530612.bdd	169	24,604	0	bbbbffff	4782ea4c	94b3de7f
32bits	SRstar(5,4,2,4)	919782430.bdd	154	24,674	0	aaaaaaaa	867fd3df	94b3de7f
32bits	SRstar(5,4,2,4)	-942445108.bdd	176	24,538	0	ffff0000	d39e7000	0000ffff
32bits	SRstar(5,4,2,4)	-734280719.bdd	201	24,184	0	0000ffff	4d158597	bbbbffff
32bits	SRstar(5,4,2,4)	590692576.bdd	119	24,67	0	174ca832	f31b7f9b	174ca832
32bits	SRstar(6,2,4,4)	1401760571.bdd	180	25,187	0	174ca832	c7d890d5	aaaaaaaa
32bits	SRstar(6,2,4,4)	-226709985.bdd	242	24,869	0	bbbbffff	07904ef6	dbc5a241
32bits	SRstar(6,2,4,4)	-407668800.bdd	59	24,059	0	aaaaaaaa	120c4c85	bbbbffff
32bits	SRstar(6,2,4,4)	931660564.bdd	225	24,395	0	ffff0000	7a641764	ffff0000
32bits	SRstar(6,2,4,4)	-158879513.bdd	61	24,079	0	dbc5a241	c0fc33f3	ffff0000
32bits	SRstar(6,2,4,4)	1412079377.bdd	246	24,859	0	0000ffff	0270395c	bbbbffff
32bits	SRstar(6,2,4,4)	211459824.bdd	58	24,091	0	aaaaaaaa	24ce4b86	ffff0000
32bits	SRstar(6,2,4,4)	-73624616.bdd	316	24,567	0	ffff0000	e3f293a3	dbc5a241
32bits	SRstar(6,2,4,4)	891512452.bdd	195	25,011	0	dbc5a241	7b976b14	dbc5a241
32bits	SRstar(6,2,4,4)	-89298611.bdd	160	24,969	0	55555555	a6326005	ffff0000
32bits	SRstar(6,2,4,4)	769598562.bdd	73	24,017	0	174ca832	7ada98be	0000ffff
32bits	SRstar(6,2,4,4)	-1830719260.bdd	128	24,881	0	55555555	4c9d8b80	55555555
32bits	SRstar(6,2,4,4)	-1867347737.bdd	137	25,125	0	174ca832	04c1f87b	174ca832
32bits	SRstar(6,2,4,4)	-2019931158.bdd	267	24,395	0	ffff0000	d37bc9b3	174ca832
32bits	SRstar(6,2,4,4)	661242879.bdd	288	24,869	0	bbbbffff	9d16acf5	55555555
32bits	SRstar(6,2,4,4)	1479373759.bdd	216	24,609	0	ffff0000	2f84dce7	0000ffff
32bits	SRstar(6,2,4,4)	465343446.bdd	246	24,6	0	bbbbffff	0a015c0e	174ca832
32bits	SRstar(6,2,4,4)	-1667000680.bdd	235	24,869	0	bbbbffff	9349d261	ffff0000
32bits	SRstar(6,2,4,4)	215228750.bdd	96	24,059	0	dbc5a241	3349c3fa	0000ffff
32bits	SRstar(6,2,4,4)	-370824289.bdd	180	24,796	0	55555555	4bfb7082	94b3de7f
32bits	SRstar(6,2,4,4)	865416163.bdd	63	24,112	0	aaaaaaaa	ece620c3	55555555
32bits	SRstar(6,2,4,4)	-1221704994.bdd	66	24,047	0	aaaaaaaa	65056ee4	0000ffff
32bits	SRstar(6,2,4,4)	-1003602085.bdd	64	24,059	0	174ca832	1bb5d2fb	dbc5a241
32bits	SRstar(6,2,4,4)	-2052099468.bdd	96	24,054	0	94b3de7f	163dc034	94b3de7f
32bits	SRstar(6,2,4,4)	842580224.bdd	189	24,974	0	94b3de7f	753d338e	ffff0000

Figure C.6: Raw data for the 32 bit systems.

32bits	SRstar(6,2,4,4)	-1189490594.bdd	256	24,616	0	ffff0000	a1a255ef	bbbbffff
32bits	SRstar(6,2,4,4)	-1751123681.bdd	253	24,869	0	bbbbffff	8f7e6946	bbbbffff
32bits	SRstar(6,2,4,4)	-2023237428.bdd	67	24,059	0	aaaaaaaa	36ab0a83	94b3de7f
32bits	SRstar(6,2,4,4)	-376016223.bdd	59	24,119	0	94b3de7f	f21f0fae	aaaaaaaa
32bits	SRstar(6,2,4,4)	-1396532786.bdd	282	24,869	0	bbbbffff	a02c8ecd	94b3de7f
32bits	SRstar(6,2,4,4)	-116723395.bdd	183	24,961	0	aaaaaaaa	b76db0c3	dbc5a241
32bits	SRstar(6,2,4,4)	1530090247.bdd	265	24,27	0	bbbbffff	04d239ef	0000ffff
32bits	SRstar(6,2,4,4)	1706152745.bdd	196	24,88	0	55555555	ccde2d15	bbbbffff
32bits	SRstar(6,2,4,4)	-2100280565.bdd	189	24,963	0	94b3de7f	e85f128d	0000ffff
32bits	SRstar(6,2,4,4)	-836283781.bdd	271	24,859	0	0000ffff	78f12cd7	174ca832
32bits	SRstar(6,2,4,4)	-1012499046.bdd	99	24,04	0	55555555	63ff09db	dbc5a241
32bits	SRstar(6,2,4,4)	-1749877345.bdd	194	24,879	0	94b3de7f	f7e31f8a	55555555
32bits	SRstar(6,2,4,4)	-283371556.bdd	247	24,395	0	ffff0000	682745ea	aaaaaaaa
32bits	SRstar(6,2,4,4)	-358066794.bdd	245	24,597	0	0000ffff	c9765082	ffff0000
32bits	SRstar(6,2,4,4)	39865149.bdd	176	25,186	0	174ca832	6fd397d1	94b3de7f
32bits	SRstar(6,2,4,4)	1755017755.bdd	243	24,646	0	ffff0000	af271e03	94b3de7f
32bits	SRstar(6,2,4,4)	2035525129.bdd	193	24,936	0	55555555	737a90af	174ca832
32bits	SRstar(6,2,4,4)	1781475304.bdd	283	24,859	0	0000ffff	1155c669	dbc5a241
32bits	SRstar(6,2,4,4)	-833341046.bdd	65	24,091	0	55555555	d17cd27f	aaaaaaaa
32bits	SRstar(6,2,4,4)	-29552625.bdd	244	24,443	0	0000ffff	b4f752d5	aaaaaaaa
32bits	SRstar(6,2,4,4)	1783239552.bdd	251	24,859	0	0000ffff	e57865a1	55555555
32bits	SRstar(6,2,4,4)	1198441733.bdd	99	24,072	0	174ca832	8dbacc72	55555555
32bits	SRstar(6,2,4,4)	14471929.bdd	254	24,859	0	0000ffff	32744d83	0000ffff
32bits	SRstar(6,2,4,4)	-500731483.bdd	60	24,059	0	dbc5a241	00b3d7bb	174ca832
32bits	SRstar(6,2,4,4)	-968905460.bdd	73	24,037	0	174ca832	44e9bbf1	bbbbffff
32bits	SRstar(6,2,4,4)	1802408517.bdd	62	24,017	0	aaaaaaaa	a38ccc8a	174ca832
32bits	SRstar(6,2,4,4)	207935197.bdd	91	24,072	0	dbc5a241	647dfc92	bbbbffff
32bits	SRstar(6,2,4,4)	1482006631.bdd	191	24,819	0	94b3de7f	30a51ca1	dbc5a241
32bits	SRstar(6,2,4,4)	1970782937.bdd	169	24,712	0	aaaaaaaa	f38d4909	aaaaaaaa
32bits	SRstar(6,2,4,4)	-42689060.bdd	135	24,83	0	dbc5a241	4bd2f8e9	aaaaaaaa
32bits	SRstar(6,2,4,4)	1254911974.bdd	92	24,069	0	55555555	77138330	0000ffff
32bits	SRstar(6,2,4,4)	-1813738253.bdd	268	24,869	0	bbbbffff	ab881739	aaaaaaaa
32bits	SRstar(6,2,4,4)	-960092673.bdd	274	24,174	0	ffff0000	6d9433a9	55555555
32bits	SRstar(6,2,4,4)	624379193.bdd	97	24,059	0	174ca832	1cad4cb0	ffff0000
32bits	SRstar(6,2,4,4)	1996626199.bdd	186	25,226	0	dbc5a241	fada7b33	94b3de7f
32bits	SRstar(6,2,4,4)	1410164745.bdd	94	24,059	0	94b3de7f	9b300519	bbbbffff
32bits	SRstar(6,2,4,4)	-1787004867.bdd	64	24,059	0	94b3de7f	a419dff3	174ca832
32bits	SRstar(6,2,4,4)	-2095202921.bdd	83	24,034	0	dbc5a241	542fc898	55555555
32bits	SRstar(6,2,4,4)	-1815920932.bdd	166	24,844	0	0000ffff	19765383	94b3de7f
32bits	SRstar(6,4,2,4)	1684385661.bdd	242	24,798	0	bbbbffff	1b7479b1	94b3de7f
32bits	SRstar(6,4,2,4)	316550913.bdd	259	24,871	0	bbbbffff	92691851	0000ffff
32bits	SRstar(6,4,2,4)	1431214749.bdd	230	24,092	0	174ca832	fad123f6	174ca832
32bits	SRstar(6,4,2,4)	45986533.bdd	247	24,822	0	55555555	ad991f32	94b3de7f
32bits	SRstar(6,4,2,4)	-267260844.bdd	241	24,823	0	dbc5a241	859f8eb5	bbbbffff
32bits	SRstar(6,4,2,4)	-1062826704.bdd	221	24,305	0	dbc5a241	ce05817e	55555555
32bits	SRstar(6,4,2,4)	1529345582.bdd	171	24,17	0	94b3de7f	ff8b1027	bbbbffff
32bits	SRstar(6,4,2,4)	1976121587.bdd	236	24,178	0	174ca832	f05e8d2d	55555555
32bits	SRstar(6,4,2,4)	336564420.bdd	266	24,237	0	bbbbffff	3a00caf6	ffff0000
32bits	SRstar(6,4,2,4)	-986409453.bdd	253	24,746	0	55555555	cca9f913	bbbbffff
32bits	SRstar(6,4,2,4)	1307993835.bdd	256	24,873	0	dbc5a241	e0657808	aaaaaaaa
32bits	SRstar(6,4,2,4)	1875749123.bdd	243	24,797	0	aaaaaaaa	63e3d6ee	ffff0000
32bits	SRstar(6,4,2,4)	-1575199319.bdd	256	24,871	0	174ca832	dfd75e92	94b3de7f
32bits	SRstar(6,4,2,4)	-1024057389.bdd	260	24,848	0	aaaaaaaa	35c5b414	94b3de7f
32bits	SRstar(6,4,2,4)	-1495853065.bdd	232	24,187	0	ffff0000	f028aea6	55555555
32bits	SRstar(6,4,2,4)	-832269232.bdd	248	24,822	0	dbc5a241	7e6a5962	174ca832
32bits	SRstar(6,4,2,4)	421167436.bdd	254	24,785	0	aaaaaaaa	8789afd3	55555555
32bits	SRstar(6,4,2,4)	-848736106.bdd	240	24,797	0	94b3de7f	ce4119ed	0000ffff
32bits	SRstar(6,4,2,4)	530302016.bdd	245	24,709	0	0000ffff	0680c2b5	55555555
32bits	SRstar(6,4,2,4)	1305275339.bdd	283	24,813	0	bbbbffff	035311b6	55555555
32bits	SRstar(6,4,2,4)	-118871252.bdd	253	24,873	0	ffff0000	c86f50a3	dbc5a241
32bits	SRstar(6,4,2,4)	1655590456.bdd	245	24,798	0	ffff0000	4f25673e	0000ffff
32bits	SRstar(6,4,2,4)	-935118922.bdd	207	24,34	0	94b3de7f	bc382310	aaaaaaaa
32bits	SRstar(6,4,2,4)	-427678663.bdd	251	24,758	0	dbc5a241	9491dae8	dbc5a241
32bits	SRstar(6,4,2,4)	892895873.bdd	248	24,797	0	55555555	278dcf9d	55555555
32bits	SRstar(6,4,2,4)	-2134145045.bdd	182	24,231	0	55555555	f2319ddd	aaaaaaaa
32bits	SRstar(6,4,2,4)	1323832391.bdd	184	24,312	0	94b3de7f	d31cac2b	dbc5a241
32bits	SRstar(6,4,2,4)	-489299267.bdd	117	24,386	0	0000ffff	a6ec832e	dbc5a241
32bits	SRstar(6,4,2,4)	-233388078.bdd	234	24,771	0	ffff0000	b3607028	94b3de7f
32bits	SRstar(6,4,2,4)	-1698952723.bdd	172	24,489	0	0000ffff	ff222d67	ffff0000
32bits	SRstar(6,4,2,4)	-619844976.bdd	252	24,822	0	174ca832	269b4d86	bbbbffff
32bits	SRstar(6,4,2,4)	1510861855.bdd	239	24,871	0	94b3de7f	b0ea26f5	ffff0000
32bits	SRstar(6,4,2,4)	-1851632015.bdd	244	24,798	0	aaaaaaaa	8cf62fca	dbc5a241
32bits	SRstar(6,4,2,4)	-659080960.bdd	258	24,797	0	aaaaaaaa	e5da384b	0000ffff

Figure C.7: Raw data for the 32 bit systems.

32bits	SRstar(6,4,2,4)	233921402.bdd	254	24,784	0	174ca832	c0b899e3	0000ffff
32bits	SRstar(6,4,2,4)	815254587.bdd	232	24,745	0	174ca832	bb2de69f	ffff0000
32bits	SRstar(6,4,2,4)	1995302341.bdd	253	24,822	0	bbbbffff	e14b4036	dbc5a241
32bits	SRstar(6,4,2,4)	-2035562895.bdd	251	24,822	0	55555555	2f575a5b	0000ffff
32bits	SRstar(6,4,2,4)	-1804307900.bdd	244	24,785	0	55555555	41a8b401	174ca832
32bits	SRstar(6,4,2,4)	269924021.bdd	244	24,798	0	ffff0000	5288201a	aaaaaaaa
32bits	SRstar(6,4,2,4)	-2130785333.bdd	243	24,798	0	aaaaaaaa	cdb366ad	aaaaaaaa
32bits	SRstar(6,4,2,4)	-1820720570.bdd	286	24,813	0	174ca832	059116ab	dbc5a241
32bits	SRstar(6,4,2,4)	-389330097.bdd	194	24,276	0	0000ffff	14a11b4a	bbbbffff
32bits	SRstar(6,4,2,4)	1431103022.bdd	227	24,216	0	ffff0000	b6fdfc03	ffff0000
32bits	SRstar(6,4,2,4)	1652204337.bdd	251	24,81	0	aaaaaaaa	6cab84e5	174ca832
32bits	SRstar(6,4,2,4)	-466722864.bdd	230	24,17	0	bbbbffff	f2758a96	bbbbffff
32bits	SRstar(6,4,2,4)	-764108127.bdd	258	24,956	0	94b3de7f	fcdddf07	174ca832
32bits	SRstar(6,4,2,4)	1191270898.bdd	256	24,848	0	dbc5a241	3908a47b	ffff0000
32bits	SRstar(6,4,2,4)	-1089493090.bdd	447	24,301	0	0000ffff	5e544d3a	174ca832
32bits	SRstar(6,4,2,4)	-1906204885.bdd	249	24,797	0	dbc5a241	2febde89	0000ffff
32bits	SRstar(6,4,2,4)	1743489814.bdd	256	24,81	0	bbbbffff	998c9107	aaaaaaaa
32bits	SRstar(6,4,2,4)	-1775971615.bdd	249	24,798	0	bbbbffff	cb8577f4	174ca832
32bits	SRstar(6,4,2,4)	-1847203766.bdd	246	24,772	0	174ca832	08955b3c	aaaaaaaa
32bits	SRstar(6,4,2,4)	-507716307.bdd	249	24,822	0	ffff0000	6e700638	bbbbffff
32bits	SRstar(6,4,2,4)	749951400.bdd	283	24,787	0	aaaaaaaa	cb45dafa	bbbbffff
32bits	SRstar(6,4,2,4)	-756199914.bdd	248	24,784	0	55555555	68d1ab91	dbc5a241
32bits	SRstar(6,4,2,4)	-1994153172.bdd	157	24,901	0	0000ffff	3f572c8b	94b3de7f
32bits	SRstar(6,4,2,4)	-555482435.bdd	258	24,773	0	94b3de7f	e2e4f50d	55555555
32bits	SRstar(6,4,2,4)	513358821.bdd	252	24,797	0	55555555	ca591192	ffff0000
32bits	SRstar(6,4,2,4)	-1509620331.bdd	258	24,785	0	ffff0000	0451aef8	174ca832
32bits	SRstar(6,4,2,4)	72421568.bdd	340	24,951	0	dbc5a241	d719ca23	94b3de7f
32bits	SRstar(6,4,2,4)	600364028.bdd	250	24,836	0	94b3de7f	2d8224f4	94b3de7f
32bits	SRstar(7,2,4,4)	-1732370947.bdd	507	24,89	0	bbbbffff	fe6876be	bbbbffff
32bits	SRstar(7,2,4,4)	253696662.bdd	353	24,879	0	94b3de7f	e9b25821	0000ffff
32bits	SRstar(7,2,4,4)	-103547209.bdd	510	24,935	0	0000ffff	af0f1a64	55555555
32bits	SRstar(7,2,4,4)	-561177221.bdd	232	24,509	0	ffff0000	a65cf53f	aaaaaaaa
32bits	SRstar(7,2,4,4)	1863109354.bdd	300	24,941	0	aaaaaaaa	08a3f1d5	0000ffff
32bits	SRstar(7,2,4,4)	-902437518.bdd	121	24,509	0	ffff0000	0a92bc12	dbc5a241
32bits	SRstar(7,2,4,4)	380485824.bdd	215	24,578	0	55555555	05afb888	55555555
32bits	SRstar(7,2,4,4)	-1369794943.bdd	349	24,889	0	94b3de7f	40a031c3	dbc5a241
32bits	SRstar(7,2,4,4)	-1189197623.bdd	352	24,935	0	94b3de7f	b305e23c	bbbbffff
32bits	SRstar(7,2,4,4)	170583520.bdd	498	24,578	0	0000ffff	0c87a393	dbc5a241
32bits	SRstar(7,2,4,4)	257385609.bdd	206	24,839	0	55555555	b2bf9fde	174ca832
32bits	SRstar(7,2,4,4)	410164843.bdd	518	24,89	0	0000ffff	c7ebe1e9	94b3de7f
32bits	SRstar(7,2,4,4)	948168745.bdd	275	24,879	0	dbc5a241	58fd143d	0000ffff
32bits	SRstar(7,2,4,4)	1650859616.bdd	316	24,908	0	174ca832	292d2cda	94b3de7f
32bits	SRstar(7,2,4,4)	274850844.bdd	217	24,941	0	55555555	5251eb5c	0000ffff
32bits	SRstar(7,2,4,4)	-1955489593.bdd	499	24,839	0	bbbbffff	d3071594	94b3de7f
32bits	SRstar(7,2,4,4)	-299039323.bdd	247	24,866	0	dbc5a241	5ca64be8	174ca832
32bits	SRstar(7,2,4,4)	-442514690.bdd	261	24,941	0	dbc5a241	cda6d581	bbbbffff
32bits	SRstar(7,2,4,4)	-681883393.bdd	291	24,833	0	aaaaaaaa	394902e9	bbbbffff
32bits	SRstar(7,2,4,4)	-1078270450.bdd	143	24,509	0	ffff0000	c95e6bdd	bbbbffff
32bits	SRstar(7,2,4,4)	626219797.bdd	503	24,822	0	0000ffff	85e30bc1	aaaaaaaa
32bits	SRstar(7,2,4,4)	-1534364330.bdd	125	24,509	0	ffff0000	20f99460	94b3de7f
32bits	SRstar(7,2,4,4)	-40395889.bdd	250	24,833	0	dbc5a241	ae1bfa61	94b3de7f
32bits	SRstar(7,2,4,4)	-554541795.bdd	285	24,942	0	174ca832	78894e83	0000ffff
32bits	SRstar(7,2,4,4)	524674573.bdd	180	24,509	0	ffff0000	d7cf7b1c	ffff0000
32bits	SRstar(7,2,4,4)	1053746098.bdd	306	24,578	0	174ca832	761dafdf	ffff0000
32bits	SRstar(7,2,4,4)	-1275556658.bdd	249	24,941	0	dbc5a241	0130bd3a	dbc5a241
32bits	SRstar(7,2,4,4)	-654362937.bdd	290	24,908	0	174ca832	933494d2	55555555
32bits	SRstar(7,2,4,4)	-1577106729.bdd	128	24,509	0	ffff0000	9003444c	0000ffff
32bits	SRstar(7,2,4,4)	-1457036776.bdd	286	24,833	0	aaaaaaaa	6750711a	dbc5a241
32bits	SRstar(7,2,4,4)	349921448.bdd	284	24,935	0	174ca832	8152efe1	aaaaaaaa
32bits	SRstar(7,2,4,4)	-1545672675.bdd	220	24,879	0	55555555	fb47d9e0	94b3de7f
32bits	SRstar(7,2,4,4)	407948565.bdd	283	24,89	0	aaaaaaaa	68da6e96	aaaaaaaa
32bits	SRstar(7,2,4,4)	-902319163.bdd	291	24,941	0	174ca832	8f593266	174ca832
32bits	SRstar(7,2,4,4)	1756073971.bdd	498	24,89	0	bbbbffff	7b2bd961	dbc5a241
32bits	SRstar(7,2,4,4)	-161037217.bdd	251	24,935	0	dbc5a241	c8f6da74	ffff0000
32bits	SRstar(7,2,4,4)	1902777045.bdd	335	24,866	0	aaaaaaaa	4dc60437	ffff0000
32bits	SRstar(7,2,4,4)	-1916970777.bdd	288	24,822	0	174ca832	c743854f	dbc5a241
32bits	SRstar(7,2,4,4)	-1420290378.bdd	198	24,833	0	55555555	6b3c1a51	dbc5a241
32bits	SRstar(7,2,4,4)	-1879413393.bdd	287	24,919	0	174ca832	f9f9ddb8	bbbbffff
32bits	SRstar(7,2,4,4)	-1955512410.bdd	519	24,89	0	0000ffff	d4c4be56	174ca832
32bits	SRstar(7,2,4,4)	1411693187.bdd	268	24,941	0	aaaaaaaa	8843b094	174ca832
32bits	SRstar(7,2,4,4)	-8586194.bdd	351	24,656	0	94b3de7f	1de1ef8f	55555555
32bits	SRstar(7,2,4,4)	724585655.bdd	264	24,926	0	dbc5a241	0eaadfc3	55555555
32bits	SRstar(7,2,4,4)	1509413657.bdd	274	24,822	0	aaaaaaaa	c82f75ff	55555555

Figure C.8: Raw data for the 32 bit systems.

32bits	SRstar(7,2,4,4)	-2102112224.bdd	348	24,879	0	94b3de7f	c4be5a73	ffff0000
32bits	SRstar(7,2,4,4)	-897517689.bdd	146	24,509	0	ffff0000	4fe68eb6	174ca832
32bits	SRstar(7,2,4,4)	-2001252215.bdd	164	24,509	0	ffff0000	18f73a3d	55555555
32bits	SRstar(7,2,4,4)	1444173511.bdd	507	24,926	0	0000ffff	a0fa610d	bbbbffff
32bits	SRstar(7,2,4,4)	-1075485366.bdd	220	24,935	0	55555555	2bd7034d	bbbbffff
32bits	SRstar(7,2,4,4)	-138505158.bdd	289	24,833	0	aaaaaaaa	cd439209	94b3de7f
32bits	SRstar(7,2,4,4)	-800547769.bdd	514	24,935	0	0000ffff	fd9eb2dc	0000ffff
32bits	SRstar(7,2,4,4)	-1962234728.bdd	215	24,578	0	55555555	ebe86a4a	aaaaaaaa
32bits	SRstar(7,2,4,4)	-1091890517.bdd	528	24,89	0	0000ffff	5733872f	ffff0000
32bits	SRstar(7,2,4,4)	1452086979.bdd	492	24,822	0	bbbbffff	5cb48ff5	55555555
32bits	SRstar(7,2,4,4)	-380204550.bdd	334	24,926	0	94b3de7f	4ae13d91	aaaaaaaa
32bits	SRstar(7,2,4,4)	-863203280.bdd	224	24,908	0	55555555	b18f2dab	ffff0000
32bits	SRstar(7,2,4,4)	-792106967.bdd	242	24,919	0	dbc5a241	be55595c	aaaaaaaa
32bits	SRstar(7,4,2,4)	2014507751.bdd	363	24,679	0	174ca832	eda6ae7c	ffff0000
32bits	SRstar(7,4,2,4)	-154587169.bdd	253	24,976	0	55555555	dbc6eda2	aaaaaaaa
32bits	SRstar(7,4,2,4)	-1258861742.bdd	494	24,618	0	0000ffff	4731249d	0000ffff
32bits	SRstar(7,4,2,4)	721789344.bdd	381	24,68	0	55555555	b2eb531f	ffff0000
32bits	SRstar(7,4,2,4)	-66154760.bdd	374	24,682	0	ffff0000	477da306	ffff0000
32bits	SRstar(7,4,2,4)	25831898.bdd	267	25,134	0	55555555	a64cde89	bbbbffff
32bits	SRstar(7,4,2,4)	-1972188431.bdd	309	24,884	0	dbc5a241	31f8c02b	bbbbffff
32bits	SRstar(7,4,2,4)	2084961481.bdd	348	24,687	0	94b3de7f	fb04af7c	dbc5a241
32bits	SRstar(7,4,2,4)	-1505925749.bdd	238	24,212	0	174ca832	560396ed	55555555
32bits	SRstar(7,4,2,4)	1891644033.bdd	392	24,784	0	aaaaaaaa	1509f756	ffff0000
32bits	SRstar(7,4,2,4)	829401083.bdd	328	24,978	0	aaaaaaaa	a2d85907	174ca832
32bits	SRstar(7,4,2,4)	-529863168.bdd	374	24,681	0	bbbbffff	424f2e92	94b3de7f
32bits	SRstar(7,4,2,4)	-812387088.bdd	201	24,858	0	ffff0000	37dc0ebe	174ca832
32bits	SRstar(7,4,2,4)	2045732254.bdd	387	24,682	0	55555555	d9e2cedb	55555555
32bits	SRstar(7,4,2,4)	-1429658701.bdd	186	24,801	0	bbbbffff	44a35b4b	ffff0000
32bits	SRstar(7,4,2,4)	1332238851.bdd	386	24,686	0	94b3de7f	37b91582	bbbbffff
32bits	SRstar(7,4,2,4)	219571760.bdd	384	24,681	0	174ca832	96effdda	dbc5a241
32bits	SRstar(7,4,2,4)	-329390060.bdd	366	24,795	0	94b3de7f	e3d7c84f	ffff0000
32bits	SRstar(7,4,2,4)	-1750814338.bdd	389	24,412	0	55555555	afe0b9b9	94b3de7f
32bits	SRstar(7,4,2,4)	1577293490.bdd	240	25,133	0	aaaaaaaa	22e76ee2	55555555
32bits	SRstar(7,4,2,4)	-1168959069.bdd	211	24,895	0	174ca832	dfa639e0	aaaaaaaa
32bits	SRstar(7,4,2,4)	382152092.bdd	467	24,651	0	0000ffff	238badcd	174ca832
32bits	SRstar(7,4,2,4)	654406926.bdd	562	24,579	0	0000ffff	0069cdb0	ffff0000
32bits	SRstar(7,4,2,4)	1885481659.bdd	210	24,52	0	94b3de7f	c2e7d58c	94b3de7f
32bits	SRstar(7,4,2,4)	-2117783417.bdd	502	24,481	0	0000ffff	b7e40abd	dbc5a241
32bits	SRstar(7,4,2,4)	258408116.bdd	111	24,553	0	174ca832	c5bc905b	0000ffff
32bits	SRstar(7,4,2,4)	-364592482.bdd	208	24,858	0	aaaaaaaa	f4682daa	aaaaaaaa
32bits	SRstar(7,4,2,4)	-898880983.bdd	412	24,683	0	55555555	07aa97fc	174ca832
32bits	SRstar(7,4,2,4)	1507137860.bdd	409	24,679	0	dbc5a241	ca334f8b	aaaaaaaa
32bits	SRstar(7,4,2,4)	-1441965982.bdd	402	24,682	0	aaaaaaaa	60ff1243	94b3de7f
32bits	SRstar(7,4,2,4)	1237794807.bdd	406	24,68	0	174ca832	1b55d86e	bbbbffff
32bits	SRstar(7,4,2,4)	-1345849789.bdd	478	24,87	0	0000ffff	adabf3d0	bbbbffff
32bits	SRstar(7,4,2,4)	-1958051187.bdd	190	24,154	0	174ca832	b3e12351	94b3de7f
32bits	SRstar(7,4,2,4)	384721544.bdd	181	24,908	0	bbbbffff	7cfd0bfb	55555555
32bits	SRstar(7,4,2,4)	221624726.bdd	392	24,681	0	ffff0000	50ef0839	aaaaaaaa
32bits	SRstar(7,4,2,4)	1640908091.bdd	404	24,418	0	94b3de7f	9f6bb7b5	0000ffff
32bits	SRstar(7,4,2,4)	-1131430779.bdd	342	24,205	0	dbc5a241	4e69ba5f	174ca832
32bits	SRstar(7,4,2,4)	292888054.bdd	301	25,132	0	174ca832	0b8584c8	174ca832
32bits	SRstar(7,4,2,4)	-1658069027.bdd	270	24,921	0	94b3de7f	afa7c4c6	55555555
32bits	SRstar(7,4,2,4)	-1820827265.bdd	393	24,678	0	ffff0000	c273d428	bbbbffff
32bits	SRstar(7,4,2,4)	-355948047.bdd	224	24,244	0	bbbbffff	8901c8a9	aaaaaaaa
32bits	SRstar(7,4,2,4)	1057638185.bdd	381	24,679	0	ffff0000	de1d6ec7	55555555
32bits	SRstar(7,4,2,4)	993981860.bdd	240	24,828	0	aaaaaaaa	0e2c364e	dbc5a241
32bits	SRstar(7,4,2,4)	687121507.bdd	380	24,683	0	bbbbffff	68b98a61	bbbbffff
32bits	SRstar(7,4,2,4)	782420330.bdd	365	24,475	0	55555555	3d593b6f	dbc5a241
32bits	SRstar(7,4,2,4)	180074381.bdd	156	24,659	0	ffff0000	e6cddd01	0000ffff
32bits	SRstar(7,4,2,4)	639063321.bdd	486	24,33	0	0000ffff	9210467d	55555555
32bits	SRstar(7,4,2,4)	1144336149.bdd	387	24,683	0	bbbbffff	8054520e	dbc5a241
32bits	SRstar(7,4,2,4)	1580785534.bdd	542	24,579	0	0000ffff	6af31fc8	aaaaaaaa
32bits	SRstar(7,4,2,4)	1055063091.bdd	248	24,52	0	94b3de7f	6cf29d16	aaaaaaaa
32bits	SRstar(7,4,2,4)	283762375.bdd	113	24,549	0	dbc5a241	09dcb353	55555555
32bits	SRstar(7,4,2,4)	700178080.bdd	270	25,132	0	dbc5a241	871393be	0000ffff
32bits	SRstar(7,4,2,4)	2002476650.bdd	388	24,693	0	dbc5a241	fe31d700	94b3de7f
32bits	SRstar(7,4,2,4)	-1113502050.bdd	152	24,659	0	ffff0000	a2237c46	dbc5a241
32bits	SRstar(7,4,2,4)	-1681142821.bdd	136	24,554	0	bbbbffff	f62deda0	174ca832
32bits	SRstar(7,4,2,4)	903147696.bdd	175	24,52	0	bbbbffff	28d100da	0000ffff
32bits	SRstar(7,4,2,4)	-875856152.bdd	254	24,976	0	ffff0000	e8461242	94b3de7f
32bits	SRstar(7,4,2,4)	-1671890382.bdd	373	24,679	0	aaaaaaaa	a267dd73	0000ffff
32bits	SRstar(7,4,2,4)	-1285372789.bdd	372	24,777	0	aaaaaaaa	b3c3053b	bbbbffff
32bits	SRstar(7,4,2,4)	-1872796640.bdd	387	24,681	0	dbc5a241	6283fb9d	ffff0000

Figure C.9: Raw data for the 32 bit systems.

32bits	SRstar(7,4,2,4)	2142960180.bdd	302	24,206	0	55555555	d1fb2962	0000ffff
32bits	SRstar(7,4,2,4)	-1690757279.bdd	111	24,52	0	dbc5a241	3ae7f46c	dbc5a241
32bits	SRstar(7,4,2,4)	-1118564095.bdd	386	24,682	0	94b3de7f	7bc298d1	174ca832
32bits	SRstar(8,2,4,4)	1821850242.bdd	483	24,287	0	bbbbffff	c0f1ca19	bbbbffff
32bits	SRstar(8,2,4,4)	-1906201711.bdd	261	24,235	0	94b3de7f	cc7042c5	dbc5a241
32bits	SRstar(8,2,4,4)	747853635.bdd	294	24,57	0	174ca832	ce04bf51	dbc5a241
32bits	SRstar(8,2,4,4)	1745091543.bdd	465	24,284	0	bbbbffff	2fd6c8d2	aaaaaaaa
32bits	SRstar(8,2,4,4)	-140817492.bdd	329	24,292	0	174ca832	df8524ca	aaaaaaaa
32bits	SRstar(8,2,4,4)	-869783632.bdd	457	24,333	0	bbbbffff	361f4dae	94b3de7f
32bits	SRstar(8,2,4,4)	905324712.bdd	483	24,859	0	94b3de7f	fc4a818e	94b3de7f
32bits	SRstar(8,2,4,4)	-944675824.bdd	322	24,691	0	dbc5a241	094a7d7b	0000ffff
32bits	SRstar(8,2,4,4)	-1226333930.bdd	548	24,682	0	bbbbffff	8725fae2	0000ffff
32bits	SRstar(8,2,4,4)	-1562273047.bdd	271	24,129	0	aaaaaaaa	c038d2b9	55555555
32bits	SRstar(8,2,4,4)	-1579531539.bdd	292	24,607	0	174ca832	cd6a5f6b	174ca832
32bits	SRstar(8,2,4,4)	1707259804.bdd	253	24,363	0	94b3de7f	d32279c2	ffff0000
32bits	SRstar(8,2,4,4)	1117302221.bdd	262	24,339	0	aaaaaaaa	e3cee93b	bbbbffff
32bits	SRstar(8,2,4,4)	272171490.bdd	535	24,406	0	ffff0000	d9bfa63c	0000ffff
32bits	SRstar(8,2,4,4)	-1183014807.bdd	315	24,643	0	aaaaaaaa	d2800381	174ca832
32bits	SRstar(8,2,4,4)	988270532.bdd	219	24,516	0	55555555	8d9a0e3f	0000ffff
32bits	SRstar(8,2,4,4)	677811583.bdd	448	24,859	0	94b3de7f	e8d537c8	aaaaaaaa
32bits	SRstar(8,2,4,4)	-580265558.bdd	268	24,6	0	94b3de7f	b5702217	bbbbffff
32bits	SRstar(8,2,4,4)	1082225281.bdd	260	24,511	0	94b3de7f	0f3fa28d	174ca832
32bits	SRstar(8,2,4,4)	887014861.bdd	148	24,15	0	bbbbffff	24ede744	ffff0000
32bits	SRstar(8,2,4,4)	955732137.bdd	335	24,425	0	dbc5a241	bbfc47d9	aaaaaaaa
32bits	SRstar(8,2,4,4)	1835892178.bdd	285	24,726	0	aaaaaaaa	be8d3475	aaaaaaaa
32bits	SRstar(8,2,4,4)	-392156674.bdd	308	24,292	0	174ca832	a153e6d2	ffff0000
32bits	SRstar(8,2,4,4)	-760912380.bdd	245	24,513	0	55555555	b4d42e2d	174ca832
32bits	SRstar(8,2,4,4)	-1167330783.bdd	456	24,284	0	bbbbffff	5f8b5d44	174ca832
32bits	SRstar(8,2,4,4)	1659419775.bdd	327	24,706	0	aaaaaaaa	e8d3b926	94b3de7f
32bits	SRstar(8,2,4,4)	-942361761.bdd	187	24,053	0	55555555	ae51f6d3	dbc5a241
32bits	SRstar(8,2,4,4)	-707387319.bdd	506	24,406	0	ffff0000	d91bc5fe	55555555
32bits	SRstar(8,2,4,4)	-1729856192.bdd	215	24,53	0	55555555	c73dff83	aaaaaaaa
32bits	SRstar(8,2,4,4)	836502902.bdd	283	24,439	0	174ca832	166c7b0d	94b3de7f
32bits	SRstar(8,2,4,4)	-1095019763.bdd	617	24,558	0	0000ffff	9f1f4d45	ffff0000
32bits	SRstar(8,2,4,4)	618396840.bdd	227	24,446	0	174ca832	827d8499	55555555
32bits	SRstar(8,2,4,4)	358486790.bdd	498	24,695	0	0000ffff	bf1a806d	55555555
32bits	SRstar(8,2,4,4)	-1934408981.bdd	465	24,284	0	0000ffff	35b0df65	94b3de7f
32bits	SRstar(8,2,4,4)	1082163650.bdd	275	24,574	0	174ca832	17d961d7	0000ffff
32bits	SRstar(8,2,4,4)	374879176.bdd	240	24,526	0	55555555	ed60c817	94b3de7f
32bits	SRstar(8,2,4,4)	616910676.bdd	457	24,591	0	0000ffff	a3f8717e	aaaaaaaa
32bits	SRstar(8,2,4,4)	2021984717.bdd	534	24,809	0	bbbbffff	5b7b28f9	dbc5a241
32bits	SRstar(8,2,4,4)	-237075364.bdd	466	24,406	0	ffff0000	bebdef64	ffff0000
32bits	SRstar(8,2,4,4)	-137591360.bdd	326	24,415	0	94b3de7f	83136ec3	0000ffff
32bits	SRstar(8,2,4,4)	916532373.bdd	474	24,859	0	ffff0000	979456a2	bbbbffff
32bits	SRstar(8,2,4,4)	-1507845935.bdd	498	24,529	0	ffff0000	72f65702	aaaaaaaa
32bits	SRstar(8,2,4,4)	-1697395288.bdd	429	24,284	0	0000ffff	21912861	0000ffff
32bits	SRstar(8,2,4,4)	98638365.bdd	226	24,285	0	dbc5a241	92ad58cc	dbc5a241
32bits	SRstar(8,2,4,4)	1387556494.bdd	346	24,775	0	aaaaaaaa	f0f07df7	0000ffff
32bits	SRstar(8,2,4,4)	653350093.bdd	499	24,668	0	ffff0000	8777cc10	174ca832
32bits	SRstar(8,2,4,4)	-1417155681.bdd	248	24,487	0	55555555	b7cc8f31	ffff0000
32bits	SRstar(8,2,4,4)	1832853168.bdd	268	24,756	0	dbc5a241	c117e517	94b3de7f
32bits	SRstar(8,2,4,4)	-2127784278.bdd	298	24,479	0	dbc5a241	81004648	174ca832
32bits	SRstar(8,2,4,4)	-1496301443.bdd	236	24,443	0	174ca832	9da207e0	bbbbffff
32bits	SRstar(8,2,4,4)	-231633935.bdd	438	24,284	0	0000ffff	49a4906a	174ca832
32bits	SRstar(8,2,4,4)	-832101668.bdd	269	24,511	0	aaaaaaaa	08db7399	dbc5a241
32bits	SRstar(8,2,4,4)	-1324811837.bdd	445	24,406	0	ffff0000	10306df7	dbc5a241
32bits	SRstar(8,2,4,4)	2047989767.bdd	234	24,285	0	dbc5a241	915d552b	55555555
32bits	SRstar(8,4,2,4)	-1270059674.bdd	693	24,087	0	0000ffff	639756ec	0000ffff
32bits	SRstar(8,4,2,4)	1104235217.bdd	676	24,087	0	0000ffff	4d9e58b0	dbc5a241
32bits	SRstar(8,4,2,4)	-1830158099.bdd	340	24,788	0	dbc5a241	3f50f50b	dbc5a241
32bits	SRstar(8,4,2,4)	-498592328.bdd	342	24,784	0	aaaaaaaa	a0ba17c3	94b3de7f
32bits	SRstar(8,4,2,4)	-1927913885.bdd	346	24,787	0	ffff0000	94d6058a	dbc5a241
32bits	SRstar(8,4,2,4)	-1855377295.bdd	324	24,909	0	ffff0000	ff566c12	0000ffff
32bits	SRstar(8,4,2,4)	-1435672254.bdd	272	24,617	0	55555555	068330ee	55555555
32bits	SRstar(8,4,2,4)	-1073921533.bdd	340	24,788	0	dbc5a241	0dffd061	55555555
32bits	SRstar(8,4,2,4)	1046435991.bdd	318	24,787	0	bbbbffff	e1356782	ffff0000
32bits	SRstar(8,4,2,4)	-152893202.bdd	354	24,786	0	55555555	6c094d8d	0000ffff
32bits	SRstar(8,4,2,4)	-2076937265.bdd	362	24,788	0	ffff0000	d3ac1001	55555555
32bits	SRstar(8,4,2,4)	-1662128825.bdd	344	24,787	0	dbc5a241	e64360d1	ffff0000
32bits	SRstar(8,4,2,4)	1804378602.bdd	339	24,788	0	94b3de7f	0bb1302f	aaaaaaaa
32bits	SRstar(8,4,2,4)	354950051.bdd	295	24,495	0	ffff0000	44697bc2	aaaaaaaa
32bits	SRstar(8,4,2,4)	308178416.bdd	352	24,787	0	dbc5a241	a5e9ede7	174ca832
32bits	SRstar(8,4,2,4)	76356391.bdd	339	24,787	0	aaaaaaaa	9d234a88	aaaaaaaa

Figure C.10: Raw data for the 32 bit systems.

32bits	SRstar(8,4,2,4)	-1805481375.bdd	337	24,787	0	aaaaaaaa	ca5b5b07	ffff0000
32bits	SRstar(8,4,2,4)	54929066.bdd	388	24,789	0	aaaaaaaa	59b596bd	55555555
32bits	SRstar(8,4,2,4)	959489004.bdd	370	24,788	0	ffff0000	6242b65f	174ca832
32bits	SRstar(8,4,2,4)	-1937470393.bdd	309	24,787	0	174ca832	ad67bbbb	55555555
32bits	SRstar(8,4,2,4)	-1921447042.bdd	191	24,128	0	dbc5a241	d1efd3cf	bbbbffff
32bits	SRstar(8,4,2,4)	-313332384.bdd	371	24,788	0	94b3de7f	3e8c7ce6	dbc5a241
32bits	SRstar(8,4,2,4)	281979822.bdd	355	24,788	0	aaaaaaaa	eac48926	bbbbffff
32bits	SRstar(8,4,2,4)	-1351536844.bdd	320	24,786	0	174ca832	fe07a233	ffff0000
32bits	SRstar(8,4,2,4)	-251177.bdd	351	24,783	0	dbc5a241	d1ac6f08	aaaaaaaa
32bits	SRstar(8,4,2,4)	-1143150769.bdd	409	24,788	0	bbbbffff	d7b0389d	aaaaaaaa
32bits	SRstar(8,4,2,4)	-1764981071.bdd	686	24,065	0	0000ffff	34451718	ffff0000
32bits	SRstar(8,4,2,4)	1881135302.bdd	333	24,785	0	174ca832	a5bf8deb	94b3de7f
32bits	SRstar(8,4,2,4)	-398356564.bdd	317	24,788	0	bbbbffff	18dfd230	0000ffff
32bits	SRstar(8,4,2,4)	-576087139.bdd	347	24,788	0	dbc5a241	df14eb72	94b3de7f
32bits	SRstar(8,4,2,4)	701844330.bdd	345	24,787	0	174ca832	d960752f	aaaaaaaa
32bits	SRstar(8,4,2,4)	1856126155.bdd	348	24,787	0	174ca832	0619d138	bbbbffff
32bits	SRstar(8,4,2,4)	133603328.bdd	367	24,786	0	ffff0000	5a8b7cfd	bbbbffff
32bits	SRstar(8,4,2,4)	-1436046827.bdd	349	24,787	0	94b3de7f	e8df2dee	55555555
32bits	SRstar(8,4,2,4)	-1321926622.bdd	303	24,788	0	bbbbffff	4ae75110	94b3de7f
32bits	SRstar(8,4,2,4)	1247303965.bdd	708	24,027	0	0000ffff	b344b5cb	aaaaaaaa
32bits	SRstar(8,4,2,4)	-1799125134.bdd	347	24,778	0	aaaaaaaa	cb0fc49a	0000ffff
32bits	SRstar(8,4,2,4)	-512023388.bdd	167	24,639	0	0000ffff	3ee90d79	bbbbffff
32bits	SRstar(8,4,2,4)	158223658.bdd	275	24,625	0	55555555	76d72ff4	dbc5a241
32bits	SRstar(8,4,2,4)	650312986.bdd	329	24,788	0	ffff0000	8ba4936a	94b3de7f
32bits	SRstar(8,4,2,4)	1188208548.bdd	672	24,087	0	0000ffff	a8d81e63	55555555
32bits	SRstar(8,4,2,4)	1627695746.bdd	379	24,825	0	174ca832	21004191	174ca832
32bits	SRstar(8,4,2,4)	-1772000405.bdd	707	24,028	0	0000ffff	0a890d7b	94b3de7f
32bits	SRstar(8,4,2,4)	-472767237.bdd	340	24,788	0	55555555	00af1815	174ca832
32bits	SRstar(8,4,2,4)	-1140787213.bdd	328	24,788	0	94b3de7f	a8bdc421	bbbbffff
32bits	SRstar(8,4,2,4)	-893829262.bdd	355	24,825	0	174ca832	aa5fdae0	0000ffff
32bits	SRstar(8,4,2,4)	609064251.bdd	707	24,088	0	0000ffff	3b7ec853	174ca832
32bits	SRstar(8,4,2,4)	1835987414.bdd	352	24,784	0	bbbbffff	497bfbec	bbbbffff
32bits	SRstar(8,4,2,4)	-2040565849.bdd	341	24,785	0	94b3de7f	d84ef1e5	174ca832
32bits	SRstar(8,4,2,4)	2131067667.bdd	244	24,469	0	ffff0000	220a7db1	ffff0000
32bits	SRstar(8,4,2,4)	-416917448.bdd	359	24,788	0	94b3de7f	c19e8b11	ffff0000
32bits	SRstar(8,4,2,4)	275985348.bdd	315	24,787	0	55555555	d80dcfd5	bbbbffff
32bits	SRstar(8,4,2,4)	413015969.bdd	388	24,819	0	55555555	d44c3476	ffff0000
32bits	SRstar(8,4,2,4)	333701419.bdd	355	24,788	0	bbbbffff	aa9569bc	174ca832
32bits	SRstar(8,4,2,4)	1445924622.bdd	330	24,787	0	55555555	97ed492b	aaaaaaaa
32bits	SRstar(8,4,2,4)	2128829668.bdd	344	24,773	0	dbc5a241	e515e72f	0000ffff
32bits	SRstar(8,4,2,4)	389105443.bdd	349	24,788	0	94b3de7f	50ab6bd1	94b3de7f
32bits	SRstar(8,4,2,4)	-648636855.bdd	343	24,788	0	174ca832	c9cff393	dbc5a241
32bits	SRstar(8,4,2,4)	1225374739.bdd	360	24,785	0	55555555	fe55705f	94b3de7f
32bits	SRstar(8,4,2,4)	-796588807.bdd	348	24,773	0	bbbbffff	7d397a08	dbc5a241
32bits	SRstar(8,4,2,4)	-261866829.bdd	366	24,784	0	94b3de7f	cb9c7876	0000ffff
32bits	SRstar(8,4,2,4)	850274087.bdd	350	24,784	0	bbbbffff	5d4eb715	55555555
32bits	SRstar(8,4,2,4)	1915846105.bdd	363	24,783	0	aaaaaaaa	4b0ffdbe	dbc5a241
32bits	SRstar(8,4,2,4)	1679826009.bdd	306	24,787	0	aaaaaaaa	554e34aa	174ca832
32bits	SRstar(9,2,4,4)	825510786.bdd	260	24,902	0	174ca832	053a155c	94b3de7f
32bits	SRstar(9,2,4,4)	-1940321493.bdd	250	24,902	0	0000ffff	dcbdc4be	0000ffff
32bits	SRstar(9,2,4,4)	-1545577278.bdd	253	24,902	0	dbc5a241	96318cc9	0000ffff
32bits	SRstar(9,2,4,4)	1843928701.bdd	242	24,902	0	bbbbffff	6a119ef2	ffff0000
32bits	SRstar(9,2,4,4)	304784830.bdd	240	24,902	0	dbc5a241	113dccd3	ffff0000
32bits	SRstar(9,2,4,4)	1895333330.bdd	255	24,902	0	bbbbffff	370c172c	bbbbffff
32bits	SRstar(9,2,4,4)	769832127.bdd	193	24,902	0	174ca832	11cd8d43	bbbbffff
32bits	SRstar(9,2,4,4)	-305649674.bdd	247	24,902	0	ffff0000	75251604	bbbbffff
32bits	SRstar(9,2,4,4)	-20189672.bdd	236	24,902	0	ffff0000	9b08594c	94b3de7f
32bits	SRstar(9,2,4,4)	-671653966.bdd	269	24,902	0	aaaaaaaa	e3b1a0ca	94b3de7f
32bits	SRstar(9,2,4,4)	-1723762928.bdd	252	24,902	0	0000ffff	8e38a24a	bbbbffff
32bits	SRstar(9,2,4,4)	1123583150.bdd	251	24,902	0	aaaaaaaa	23bc7d69	55555555
32bits	SRstar(9,2,4,4)	-967128562.bdd	258	24,902	0	ffff0000	d39e177f	dbc5a241
32bits	SRstar(9,2,4,4)	1028416806.bdd	278	24,902	0	aaaaaaaa	9816acc7	0000ffff
32bits	SRstar(9,2,4,4)	47403550.bdd	238	24,902	0	174ca832	3da36dfb	55555555
32bits	SRstar(9,2,4,4)	1269302248.bdd	333	24,902	0	ffff0000	726a77e7	ffff0000
32bits	SRstar(9,2,4,4)	646307705.bdd	234	24,902	0	94b3de7f	94a6a0df	dbc5a241
32bits	SRstar(9,2,4,4)	951867644.bdd	208	24,902	0	94b3de7f	2abc851d	bbbbffff
32bits	SRstar(9,2,4,4)	-851021731.bdd	269	24,902	0	dbc5a241	3b6bf799	174ca832
32bits	SRstar(9,2,4,4)	910259526.bdd	205	24,902	0	94b3de7f	416460bc	0000ffff
32bits	SRstar(9,2,4,4)	-482549620.bdd	246	24,902	0	dbc5a241	dda25499	dbc5a241
32bits	SRstar(9,2,4,4)	-1395823806.bdd	239	24,902	0	ffff0000	1fee7918	55555555
32bits	SRstar(9,2,4,4)	-1794449643.bdd	202	24,902	0	94b3de7f	d4f87569	94b3de7f
32bits	SRstar(9,2,4,4)	1285551990.bdd	280	24,902	0	174ca832	a1a9b252	aaaaaaaa
32bits	SRstar(9,2,4,4)	-979445336.bdd	243	24,902	0	94b3de7f	52e0f4db	aaaaaaaa

Figure C.11: Raw data for the 32 bit systems.

32bits	SRstar(9,2,4,4)	1513773878.bdd	195	24,902	0	94b3de7f	c22575bc	174ca832
32bits	SRstar(9,2,4,4)	-2135651762.bdd	208	24,902	0	94b3de7f	e01dc003	55555555
32bits	SRstar(9,2,4,4)	-1055223280.bdd	237	24,902	0	bbbbffff	76b33122	dbc5a241
32bits	SRstar(9,2,4,4)	-646225362.bdd	243	24,902	0	dbc5a241	02ceb284	bbbbffff
32bits	SRstar(9,2,4,4)	-629077673.bdd	243	24,902	0	0000ffff	2b26c56d	174ca832
32bits	SRstar(9,2,4,4)	922013425.bdd	247	24,902	0	bbbbffff	a6c78a30	aaaaaaaa
32bits	SRstar(9,2,4,4)	1256152527.bdd	250	24,902	0	bbbbffff	b4cfaf02	174ca832
32bits	SRstar(9,2,4,4)	2080706327.bdd	230	24,902	0	dbc5a241	714774e2	55555555
32bits	SRstar(9,2,4,4)	-908469147.bdd	251	24,902	0	0000ffff	9e88e5dc	ffff0000
32bits	SRstar(9,2,4,4)	1671291855.bdd	239	24,902	0	aaaaaaaa	5a099a2f	bbbbffff
32bits	SRstar(9,2,4,4)	156172800.bdd	231	24,902	0	174ca832	3f0abef5	174ca832
32bits	SRstar(9,2,4,4)	-979822164.bdd	231	24,902	0	dbc5a241	61c84963	aaaaaaaa
32bits	SRstar(9,2,4,4)	-1614937213.bdd	235	24,902	0	dbc5a241	257ad592	94b3de7f
32bits	SRstar(9,2,4,4)	1046931065.bdd	215	24,902	0	94b3de7f	a54bbe20	ffff0000
32bits	SRstar(9,2,4,4)	-1894924141.bdd	245	24,902	0	bbbbffff	cea9563e	55555555
32bits	SRstar(9,2,4,4)	851441560.bdd	259	24,902	0	0000ffff	33c0b1d0	aaaaaaaa
32bits	SRstar(9,2,4,4)	1603439472.bdd	285	24,902	0	aaaaaaaa	6f592900	174ca832
32bits	SRstar(9,4,2,4)	1496602286.bdd	588	24,529	0	bbbbffff	3df762e5	94b3de7f
32bits	SRstar(9,4,2,4)	-915377699.bdd	450	24,649	0	ffff0000	d176bfc4	dbc5a241
32bits	SRstar(9,4,2,4)	1393679835.bdd	548	24,613	0	aaaaaaaa	9cda9ef6	dbc5a241
32bits	SRstar(9,4,2,4)	1691906919.bdd	294	24,11	0	ffff0000	7fb51af8	174ca832
32bits	SRstar(9,4,2,4)	595485540.bdd	535	24,67	0	55555555	047ab187	aaaaaaaa
32bits	SRstar(9,4,2,4)	-994087749.bdd	529	24,374	0	dbc5a241	786009ed	aaaaaaaa
32bits	SRstar(9,4,2,4)	670929749.bdd	433	24,119	0	bbbbffff	a0902159	dbc5a241
32bits	SRstar(9,4,2,4)	842582892.bdd	411	24,053	0	bbbbffff	51979586	55555555
32bits	SRstar(9,4,2,4)	-1266068672.bdd	414	24,053	0	bbbbffff	642c0c31	bbbbffff
32bits	SRstar(9,4,2,4)	-349936679.bdd	457	24,087	0	aaaaaaaa	235e20a5	94b3de7f
32bits	SRstar(9,4,2,4)	-818476290.bdd	514	24,324	0	174ca832	b63507d4	0000ffff
32bits	SRstar(9,4,2,4)	1676866962.bdd	489	24,211	0	dbc5a241	6dce956a	174ca832
32bits	SRstar(9,4,2,4)	816891288.bdd	621	24,571	0	174ca832	d0074114	ffff0000
32bits	SRstar(9,4,2,4)	1689388240.bdd	513	24,359	0	55555555	8ed41d99	ffff0000
32bits	SRstar(9,4,2,4)	334423831.bdd	534	24,336	0	aaaaaaaa	e325580e	bbbbffff
32bits	SRstar(9,4,2,4)	-1186882066.bdd	383	24,571	0	0000ffff	e0c1da84	aaaaaaaa
32bits	SRstar(9,4,2,4)	-592542581.bdd	444	24,031	0	94b3de7f	c0819131	174ca832
32bits	SRstar(9,4,2,4)	63507155.bdd	582	24,615	0	55555555	7a2056c2	174ca832
32bits	SRstar(9,4,2,4)	1244674048.bdd	493	24,215	0	dbc5a241	c2855090	0000ffff
32bits	SRstar(9,4,2,4)	-2016095035.bdd	501	24,297	0	94b3de7f	1e01cadf	dbc5a241
32bits	SRstar(9,4,2,4)	1443932465.bdd	309	24,318	0	0000ffff	81a5a55c	ffff0000
32bits	SRstar(9,4,2,4)	348351223.bdd	422	24,134	0	174ca832	e7226e0a	dbc5a241
32bits	SRstar(9,4,2,4)	138710993.bdd	542	24,676	0	94b3de7f	990508ca	55555555
32bits	SRstar(9,4,2,4)	-1394910365.bdd	575	24,802	0	94b3de7f	7d98eba1	0000ffff
32bits	SRstar(9,4,2,4)	9244328.bdd	561	24,317	0	aaaaaaaa	79a9ec11	55555555
32bits	SRstar(9,4,2,4)	-797694971.bdd	320	24,143	0	ffff0000	8be9eccb	55555555
32bits	SRstar(9,4,2,4)	360984452.bdd	593	24,48	0	dbc5a241	1637698d	dbc5a241
32bits	SRstar(9,4,2,4)	-2028596508.bdd	498	24,198	0	94b3de7f	b9302732	bbbbffff
32bits	SRstar(9,4,2,4)	-514912476.bdd	383	24,051	0	bbbbffff	4a692cfa	ffff0000
32bits	SRstar(9,4,2,4)	-1311110028.bdd	385	24,071	0	aaaaaaaa	32924ead	ffff0000
32bits	SRstar(9,4,2,4)	-583921562.bdd	542	24,864	0	174ca832	5d752d28	aaaaaaaa
32bits	SRstar(9,4,2,4)	-18936451.bdd	534	24,613	0	55555555	ae300ce0	94b3de7f
32bits	SRstar(9,4,2,4)	255472652.bdd	499	24,191	0	174ca832	19aa48c2	94b3de7f
32bits	SRstar(9,4,2,4)	-277338961.bdd	525	24,324	0	55555555	4d2352cf	bbbbffff
32bits	SRstar(9,4,2,4)	263624275.bdd	573	24,318	0	94b3de7f	77bb2ca9	ffff0000
32bits	SRstar(9,4,2,4)	-1633352441.bdd	388	24,053	0	94b3de7f	a65bcc6e	94b3de7f
32bits	SRstar(9,4,2,4)	-2144976975.bdd	522	24,672	0	55555555	4cbcff43	0000ffff
32bits	SRstar(9,4,2,4)	-998471127.bdd	449	25,106	0	ffff0000	92ed6665	aaaaaaaa
32bits	SRstar(9,4,2,4)	-610321795.bdd	298	24,11	0	ffff0000	ab96c0a5	94b3de7f
32bits	SRstar(9,4,2,4)	-184267299.bdd	573	24,571	0	0000ffff	e1af1a79	bbbbffff
32bits	SRstar(9,4,2,4)	1653175836.bdd	319	24,03	0	ffff0000	70bc7b87	bbbbffff
32bits	SRstar(9,4,2,4)	-1109402089.bdd	465	24,668	0	55555555	c30868e2	dbc5a241
32bits	SRstar(9,4,2,4)	-1683569531.bdd	420	24,128	0	dbc5a241	2818635d	bbbbffff
32bits	SRstar(9,4,2,4)	794962674.bdd	563	24,704	0	bbbbffff	c5a471b1	0000ffff
32bits	SRstar(9,4,2,4)	-1169724452.bdd	505	24,199	0	174ca832	eaad76b3	55555555
32bits	SRstar(9,4,2,4)	211354408.bdd	409	24,571	0	0000ffff	1276ec49	174ca832
32bits	SRstar(9,4,2,4)	-1213242331.bdd	522	24,248	0	aaaaaaaa	7b7d8539	0000ffff
32bits	SRstar(9,4,2,4)	-206783242.bdd	481	24,872	0	ffff0000	a5df90fc	ffff0000
32bits	SRstar(9,4,2,4)	1662847671.bdd	442	24,053	0	dbc5a241	506cbadd	ffff0000
32bits	SRstar(9,4,2,4)	2035125931.bdd	448	24,571	0	0000ffff	d71f8b32	94b3de7f
32bits	SRstar(9,4,2,4)	-1664500771.bdd	706	24,526	0	aaaaaaaa	c8b35596	aaaaaaaa
32bits	SRstar(9,4,2,4)	-566805912.bdd	490	24,157	0	bbbbffff	3853ee66	174ca832
32bits	SRstar(9,4,2,4)	-640006913.bdd	282	24,06	0	dbc5a241	93e5496b	94b3de7f
32bits	SRstar(9,4,2,4)	-1752858083.bdd	423	24,054	0	aaaaaaaa	63c2cd89	174ca832
32bits	SRstar(9,4,2,4)	499145039.bdd	521	24,162	0	174ca832	19528978	174ca832
32bits	SRstar(9,4,2,4)	391630247.bdd	549	24,375	0	bbbbffff	3bc3060d	aaaaaaaa

Figure C.12: Raw data for the 32 bit systems.