An Algebraic Attack on Block Ciphers

by

Kenneth Matheis

A Dissertation Submitted to the Faculty of

The Charles E. Schmidt College of Science

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

Florida Atlantic University

Boca Raton, Florida

December 2010

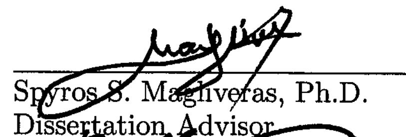
An Algebraic Attack on Block Ciphers

by

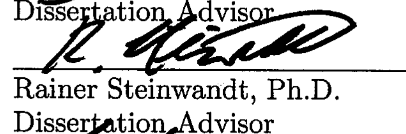
Kenneth Matheis

This dissertation was prepared under the direction of the candidate's dissertation advisors, Dr. Spyros Magliveras and Dr. Rainer Steinwandt of the Department of Mathematical Sciences, and it has been approved by the members of his supervisory committee. It was submitted to the faculty of the Charles E. Schmidt College of Science and was accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

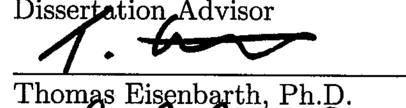
SUPERVISORY COMMITTEE:



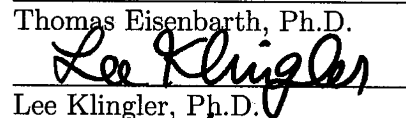
Spyros S. Magliveras, Ph.D.
Dissertation Advisor



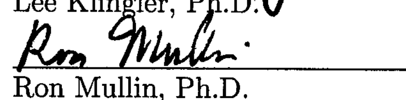
Rainer Steinwandt, Ph.D.
Dissertation Advisor



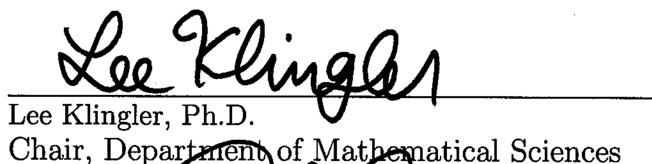
Thomas Eisenbarth, Ph.D.



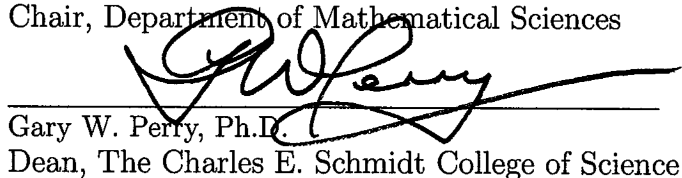
Lee Klingler, Ph.D.



Ron Mullin, Ph.D.



Lee Klingler, Ph.D.
Chair, Department of Mathematical Sciences



Gary W. Perry, Ph.D.
Dean, The Charles E. Schmidt College of Science

Barry T. Rosson, Ph.D.
Dean, Graduate College

Date

Acknowledgments

First, I would like to express my deepest gratitude to all members of my dissertation committee, Professors Spyros S. Magliveras, Rainer Steinwandt, Thomas Eisenbarth, Lee Klingler, and Ron Mullin. I appreciate their time, interest, support, and valuable comments.

For the last five years, I have had the opportunity and the privilege to work with Dr. Magliveras. Under his guidance I developed a strong sense of connection between different branches of discrete mathematics and a love of all theories Coding and Group. I am thankful to him for allowing me to forge my own research path, and for caring for me during my darkest hour.

For the last three years, I have also had the opportunity and the privilege to work with Dr. Steinwandt. Under his guidance I learned a great many things about the state of cryptography and the academic process, and his input has been invaluable. His impulsive words "I want to write a hardware paper" gave birth to PET SNAKE, and I cherish the many conversations we've had on this and other topics.

Though Dr. Eisenbarth is relatively new to my committee, and I was his first advisee since he came to Florida Atlantic University, he has been wonderful as a sounding board and as a source of ideas (and ciphers!). This body of work would undoubtedly be less substantial without his input over the last year.

Dr. Klingler has been nothing but supportive through my studies, both as an instructor and as a department head, and I thank him for his gentle encouragement

to solve tough problems as well as his willingness to listen to whatever ideas came into my head on any given morning.

Further, I would not have entered this Ph.D. program without the support of Dr. Mullin, who taught my very first course on cryptography, and who explained each topic therein with a certain conciseness I can only hope to emulate. I regret not being able to take more courses with him, cryptography-related or otherwise.

I also owe thanks to Willi Geiselmann, an excellent individual whose work on PET SNAKE was of supreme help. Working with him and Rainer on PET SNAKE is truly one of the best collaborative experiences I've ever had.

Certainly this work would not be possible without help from the Center for Cryptology and Information Security at Florida Atlantic University, and so I give it (and its members) thanks.

I owe special thanks to Dr. Stephen Locke and Dr. Heinrich Niederhausen for their tremendous help and support in many ways, from the very first day I arrived at Florida Atlantic University. Their sense of humor often had kept me going when my Pepsi did not.

I also give thanks to my instructors at the Institute for Mathematics and Computer Science (notably Dr. Iain Ferguson, Dr. Ed Martin, and Dr. Burt Kaufman) for their wonderful, beautiful program I was lucky enough to participate in through middle and high school. I blame them for my love of (and sense of) abstraction. I also thank Terry Kaufman, IMACS President, and Dr. Ted Sweet, IMACS Director, for keeping me employed through this effort.

Finally, I am grateful to my blood family and chosen family for their love and constant support through this journey. Their presence has helped me through more than one difficult period of my life, and without them this work would not be possible. Sylvia, thank you for waiting for me.

Abstract

Author: Kenneth Matheis

Title: An Algebraic Attack on Block Ciphers

Institution: Florida Atlantic University

Dissertation Advisors: Dr. Spyros S. Magliveras and Dr. Rainer Steinwandt

Degree: Doctor of Philosophy

Year: 2010

The aim of this work is to investigate an algebraic attack on block ciphers called Multiple Right Hand Sides (MRHS). MRHS models a block cipher as a system of n matrix equations $S_i := A_i x = [L_i]$, where each L_i can be expressed as a set of its columns $b_{i1}, \ldots, b_{is_i}$. The set of solutions T_i of S_i is defined as the union of the solutions of $A_i x = b_{ij}$, and the set of solutions of the system $S_1, \ldots, S_n$ is defined as the intersection of $T_1, \ldots, T_n$. Our main contribution is a hardware platform which implements a particular algorithm that solves MRHS systems (and hence block ciphers). The case is made that the platform performs several thousand orders of magnitude faster than software, it costs less than US$1,000,000, and that actual times of block cipher breakage can be calculated once it is known how the corresponding software behaves. Options in MRHS are also explored with a view to increase its efficiency.

Dedication

This work is dedicated to my mother, who sacrificed entirely too much so that I could have a chance, and to my father, who always stands beside me.

Contents

List of Tables x

List of Figures xiii

1 Introduction 1

2 MRHS 5

2.1 Preliminaries 5

2.2 Solving a System of Symbols 6

2.2.1 Agreeing 7

2.2.2 Gluing 10

2.2.3 Gluing Methods 11

2.2.4 Extracting Equations 13

2.2.5 The Equation Symbol 14

2.2.6 Guessing Variables 14

2.2.7 Guessing Methods 15

2.2.8 Putting it All Together 17

3 Software 22

3.1 An Explosion of Options 23

3.1.1 Cipher Choice and Rounds 23

3.1.2 Multiple Plaintext/Ciphertext Pairs 25

3.1.3 Other Options 27
3.2 Software Attacks 28
3.2.1 Brute Force 29
3.2.2 Discovering δ 30
3.2.3 Brute Force, Revisited 39
3.2.4 Flow Variations 42
3.2.5 Guessing Variations 43
3.2.6 Glue Variations 44
3.2.7 Key Variations 45
3.2.8 Pair Variations 46

4 Hardware 51
4.1 Overall Architecture 54
4.1.1 Initialization 55
4.1.2 Processing of Symbols 56
4.1.3 PET SNAKE's Agreement Phase 58
4.1.4 PET SNAKE's Equation Propagation Phase 62
4.1.5 PET SNAKE's Glue Phase 64
4.1.6 Parallelism 66
4.2 Main Processing Unit 66
4.2.1 MPU Data Flow 66
4.2.2 Traffic Controller 72
4.2.3 Row Reducer 72
4.2.4 Multiplier 73
4.2.5 Hash Table 73
4.2.6 Adder 74

4.3 Performance I: Total Chip Area and Cost 74
4.4 Performance II: PET SNAKE vs. Software 75
4.4.1 Linear Algebra 76
4.4.2 Matrix Multiplication and Recording Deletions 79
4.4.3 Gluing 81
4.4.4 Equation Extraction 82
4.4.5 Software Measurement 85
4.5 Performance III: Parallelization 88

5 The Transfer Formula 91
5.1 PET SNAKE's Guess Tree 91
5.2 The Deletion Model 94
5.2.1 Rooted Trees 94
5.2.2 Generating Rooted Forests Uniformly at Random 96
5.2.3 Experiments on Rooted Forests 104
5.3 The Transfer Formula 106
5.3.1 Multipliers 107
5.3.2 Turn Estimates 108
5.3.3 Evaluating Total Cost 114
5.4 Some Examples 115
5.4.1 AES 115
5.4.2 DESL 116
5.4.3 PRESENT 116
5.4.4 KeeLoq 118
5.4.5 KATAN 118

6 Conclusion 119

6.1 The Present 119
6.2 The Future 121
6.2.1 Statistics 121
6.2.2 Software 121
6.2.3 Hardware 122

A Traffic Controller 124

B Row Reducer 128

C Multiplier 138

D Hash Table 143

E Adder 151

F ASIC Implementation Details 155

Bibliography 157

List of Tables

2.1 Bits to Guess in Clump Search After First Turn 16

3.1 Computers Used in Execution 28

3.2 Number of Attacks on 4 Rounds of PRESENT Requiring Listed Guessed Bits to Determine Inconsistency on Ares-type Machines . . 32

3.3 AES δ and Attack Runtimes on Blue, Varying Rounds 33

3.4 AES Attack Runtimes on Blue as Percentages of Total Time 33

3.5 AES δ on Blue Using Linear Choice, Varying Rounds and Threshold 34

3.6 DESL δ on Blue Using Linear Choice, Varying Rounds and Threshold 35

3.7 PRESENT δ on Pink Using Linear Choice, Varying Rounds and Threshold 36

3.8 KeeLoq δ on Blue Using Linear Choice, Varying Rounds and Threshold 37

3.9 KeeLoq δ on Blue Using Linear Choice, Varying Rounds 37

3.10 KATAN δ on Pink Using Clump Search, Small Rounds 38

3.11 KATAN δ on Pink Using Clump Search, Large Rounds 38

3.12 Estimated Brute Force and MRHS Depth-First Search Times 41

3.13 PRESENT δ and Time on Ritsuko, Varying Rounds and Flow . . . 42

3.14 DESL δ on Blue Using Hot Bit Choice, Varying Rounds and Threshold 43

3.15 4 Rounds of PRESENT δ on Pink, Varying Threshold and Guessing Method 43

3.16 PRESENT and DESL δ on Aoba, Varying Rounds and Gluing Method 44
3.17 PRESENT δ on Yui and Aoba, Varying Rounds and Key Choice . . 45
3.18 4 Rounds of PRESENT δ with Distributed Method on Ares-type Machines 46
3.19 AES δ on Pink, Varying Pairs 47
3.20 DESL δ and Time on Blue, Varying Pairs 48
3.21 4 Rounds of PRESENT δ, Varying Pairs via Increment Way 48
3.22 4 Rounds of PRESENT δ, Varying Pairs via Rotation Way 49
3.23 KATAN δ, Varying Threshold 50

4.1 Size of Individual MPU Components 75
4.2 Some Measured Values of Software Performance ($k = 8$, $\beta = 32$, $\gamma = 0.5$, $y = 308$) 87
4.3 Some Projected Values of Software Performance ($k = 8$, $\beta = 32$, $\gamma = 0.5$, $y = 308$, $\alpha = 66.068$) 89

5.1 Bits to Guess After Base State is Reached 92
5.2 Experimental Procedure Results ($s = t = 1000$) 105
5.3 PET SNAKE Estimated Runtimes for AES, Varying Rounds 115
5.4 PET SNAKE Estimated Runtimes for DESL, Varying Rounds . . . 116
5.5 PET SNAKE Estimated Runtimes for PRESENT, Varying Rounds and Key Choice 117
5.6 PET SNAKE Estimated Runtimes for 4 Rounds of PRESENT, Varying Pairs 117
5.7 PET SNAKE Estimated Runtimes for KeeLoq, Varying Rounds . . 118
5.8 PET SNAKE Estimated Runtimes for KATAN, Varying Rounds . . 118

F.1 Transistor Counts of Logic Gates and Components 155

F.2 Average Surface Area of Hardware Components 155

List of Figures

2.1 Agreeing Two Symbols $A_i x = [L_i]$ and $A_j x = [L_j]$, where $L_\eta \in \mathbb{F}_2^{k_\eta \times s_\eta}$ 8

2.2 Agreeing1 Algorithm 8

2.3 Basic Flow 18

2.4 Schoonen Flow 19

2.5 PET SNAKE Flow 21

4.1 Overall architecture of PET SNAKE 55

4.2 Overall algorithm run by PET SNAKE 57

4.3 MPU Busing Diagram (High Level) 67

4.4 High Level Order of Operations During an Agreement 68

4.5 High Level Order of Operations During Gluing 69

4.6 High Level Order of Operations of Extracting Equations from a Symbol 70

4.7 High Level Order of Operations During a Mass Row Reduction . . . 71

5.1 Experimental Procedure 105

B.1 JONES Element (High Level) 129

B.2 JONES Element (Low Level) 130

B.3 Processing Algorithm 132

B.4 Rotating `U` into Proper Position 133

C.1 `Ur` and `Us` (Low Level) 139

D.1 Hash Table Entry for a Value r 144

Chapter 1

Introduction

Cryptography is a broad discipline encompassing the ideas of hiding information, establishing protocols to transfer information, securing information delivery, and verifying the authenticity of information received. If we wish to transmit a message securely, it is often called the *plaintext*, and the text that results from obscuring the plaintext is called the *ciphertext*. The earliest known cryptographic practices include transposing letters of the plaintext (giving rise to *transposition ciphers*) and substituting letters of the plaintext (giving rise to *substitution ciphers*). A substitution cipher was employed by Julius Caesar to communicate with his generals during military campaigns. These methods are primitive by today's standards, for one needs only to perform a frequency analysis of the ciphertext to discover the plaintext.

In situations where it is not feasible to have visible ciphertext, *steganographic* methods were developed. There are a variety of implementations of these methods. An early example of this is hiding a message as a tattoo on a slave's shaved head, where the regrown hair serves as the hiding agent [20]. In letters and print media, one may perform *letter hiding*. In letter hiding, the plaintext is hidden in plain sight scattered among other letters, and the recipient is made aware of where to

look, such as the first letter of a sentence written in a newspaper, or at predefined positions in a private (but eavesdropped) communication. More modern techniques include invisible ink and microdots.

A fundamental weakness of the above methods is that knowledge of the process itself leads an attacker to the way to discover the plaintext. One of the first ciphers that showed some security even if the process was known was the Vigenère cipher, which encrypts each letter of the plaintext with a different substitution cipher. To determine which substitution was used on which letters, a word or phrase called the *key* was chosen; each letter of the key determined which substitution to perform on the corresponding plaintext character.

It was shown in the mid 1800s that ciphers of this type were partially vulnerable to extended frequency analysis techniques [20], but key-dependent cryptography, as an idea, remained. A few decades later Kerckhoffs's Principle arose: A cryptosystem should be secure even if everything about the system, except the key, is public knowledge [21]. Claude Shannon, often referred to as the father of information theory, later restated this principle as "the enemy knows the system", and this restatement is known as Shannon's Maxim. He also theorized a method of encryption where both parties use a pre-shared secret for both encryption and decryption [39].

This spurned the theoretical development and robust implementations of what is known as private-key and public-key cryptography. In private-key cryptography, the same key is used to encrypt and decrypt a message, but in public-key cryptography, two keys are used: one for encryption (which is made freely available), and one for decryption (which is kept secret).

We largely focus on the fruits of private-key cryptography in the digital age, most notably ciphers like DES [28], AES [30], and PRESENT [4]. In such ciphers, the plaintext (rendered as a bitstring) is broken up into blocks, and each block is

encrypted with a key of some number of bits. (Hence, such ciphers are called *block ciphers*.) The encryption process involves substituting small blocks of data for other small blocks by way of an *S-box*, and permuting bits from one position to another. The key is often transformed as well, and after a substitution and a permutation of the plaintext has occurred, the result is typically added to some data derived from the key. Since this occurs at the bit level, often this addition is performed over $\mathbb{F}_2$, which is equivalent to XORing the bits together. This sequence of events is usually called a *round* of encryption, and the full process of encryption takes several rounds.

To attack these kinds of ciphers, the techniques of linear cryptanalysis and differential cryptanalysis were developed, among others. Many cipher designs have fallen victim to these attacks, and subsequently they do not enjoy popularity today. Of those that remain, the question naturally arose as to how to best go about breaking them. Thus was born the technique of algebraic cryptanalysis. In this technique, the cipher is instead modelled as a series of polynomial equations, and the aim is to solve the system for the variables representing the key. An implementation of such a solution is called an *algebraic attack*. Such an idea had been theorized by Shannon in 1949 in his famous quote "if we could show that solving a certain system requires at least as much work as solving a system of simultaneous equations in a large number of unknowns, of a complex type, then we would have a lower bound of sorts for the work characteristic" [39], but actual implementation of such models occurred much later. One such implementation for AES, for example, can be found in [27].

Algebraic attacks have become an important cryptanalytic tool, and the security of a cryptographic algorithm relies on the infeasibility of solving its associated system of polynomial equations. Popular attacks are based on the use of Gröbner basis techniques and SAT-solvers—prominent examples including Buchmann et al.'s

discussion of AES-128 [9] and Courtois et al.'s discussion of KeeLoq [11]. Adding to the toolbox of algebraic cryptanalysis, in [32] Raddum and Semaev propose a technique known as *MRHS* (**M**ultiple **R**ight **H**and **S**ides) to handle polynomial systems of equations over $\mathbb{F}_2$. This algorithm is particulary well-suited for describing systems of equations for an algebraic key recovery attack against common, popular block ciphers. It is this algorithm that this work primarily focuses on.

Chapter 2 details the MRHS algorithm, its components, and ways those can be put together to break a block cipher. Chapter 3 discusses the software methods of implementation, the variations therein, and the data we extract from them when applied to some block ciphers, making the case for why different options in the technique are preferable. Chapter 4 provides in detail a hardware implementation of MRHS called *PET SNAKE* (short for **P**arallel **E**limination **T**echnique **S**upporting **N**ice **A**lgebraic **K**ey **E**limination) and establishes its components' runtime gains over the appropriate components in software. Chapter 5 discusses how to take data generated from a software run of an attack to predict the actual running time of PET SNAKE, putting together the hardware gains theorized in the previous chapter. Finally, Chapter 6 gives the strides this work contributes to the community and ideas for future work along this line.

Chapter 2

MRHS

An MRHS attack is most commonly performed when the attacker has one plaintext/ciphertext pair already in hand [34, p. 70]. The attack is used to discover the key. All software experiments prior to this work have followed this model. One of the contributions of this work is the method of constructing an MRHS attack using multiple plaintext/ciphertext pairs, but this discussion is postponed until Chapter 3.

2.1 Preliminaries

We define $\mathbb{W}$ to be the set of whole numbers $\{0, 1, 2, \dots\}$. We define $\mathbb{N}$ to be the set of natural numbers $\{1, 2, 3, \dots\} = \mathbb{W} - \{0\}$. For each $n \in \mathbb{N}$, we define the set seg_n to be $\{1, 2, 3, \dots, n\}$. Finally, for any finite set A, we denote its number of elements by $\#A$.

Let $x = (x_1\ x_2\ \cdots\ x_y)^{\mathrm{T}}$ be a column vector consisting of y Boolean variables, let A be a $k \times y$ binary matrix of rank k, and let $b_1, b_2, \dots, b_s$ be column vectors of length k. An equation

$$Ax = b_1, b_2, \dots, b_s \tag{2.1.1}$$

is called an *MRHS system of linear equations* with *right hand sides* $b_1, b_2, \dots, b_s$. A

solution to (2.1.1) is a vector in $\mathbb{F}_2^y$ satisfying one of the particular linear equation systems $Ax = b_i$. The set of *all solutions to* (2.1.1) is the union of solutions to the individual linear systems $Ax = b_i$ $(i \in seg_s)$. In an effort to manipulate the data contained in the above column vectors b_i, we write them side-by-side to form a matrix L and rewrite equation (2.1.1) as $Ax = [L]$. The brackets around L emphasize that we are not working with a regular equation of matrices, and instead of an *MRHS system of linear equations*; the term *symbol* is often used to refer to the pair (A, L).

Given a system of symbols

$$\begin{array}{rrcl} S_1 : & A_1 x & = & [L_1] \\ & & \vdots & , \\ S_n : & A_n x & = & [L_n] \end{array} \qquad (2.1.2)$$

by a *solution to such a system* we mean a vector in $\mathbb{F}_2^y$ satisfying all of the underlying n MRHS systems of linear equations (where $x = (x_1\ x_2\ \cdots\ x_y)^{\mathrm{T}}$). The goal of the algorithm discussed next is to find all solutions of (2.1.2). The process of creating a system of symbols from a cryptosystem is discussed nicely in [34, Chapter 5].

2.2 Solving a System of Symbols

There are three main components to MRHS which are called *agreeing*, *gluing*, and *extracting equations*. Since memory is finite in any implementation, we may also have to *guess variables* on occasion. Further, the use of an *equation symbol* is sometimes employed. Each of these is discussed below.

2.2.1 Agreeing

The basic approach is to remove some of the columns b in a right hand side L_i, if no one solution of $A_i x = b$ can be a solution to the system (2.1.2). The mechanism by which this is achieved is pairwise *agreeing* of symbols. Namely, let $S_i : A_i x = [L_i]$ and $S_j : A_j x = [L_j]$ be two symbols. Then S_i and S_j *agree* if for every $b \in L_i$, there exists a $b' \in L_j$ such that the linear system

$$\begin{pmatrix} A_i \\ A_j \end{pmatrix} x = \begin{pmatrix} b \\ b' \end{pmatrix} \tag{2.2.1}$$

is consistent, and, vice versa, for each $b' \in L_j$ there exists a $b \in L_i$ such that (2.2.1) is consistent.

When S_i and S_j do not agree, one removes those columns b from L_i for which the linear system $A_i x = b$ is inconsistent with $A_j x = [L_j]$. Dually, those columns b' from L_j are removed, for which $A_j x = b'$ is inconsistent with $A_i x = [L_i]$. Different strategies can be used for this approach.

Let $t = k_i + k_j$ and let $A = \binom{A_i}{A_j}$ be the vertical concatenation of A_i and A_j, i. e., A has t rows. Let $T_{ij} = \binom{L_i}{0}$ and $T_{ji} = \binom{0}{L_j}$. Both T_{ij} and T_{ji} have t rows each. We follow the technique in Figure 2.1 (see [32, Section 3]) and realize it both in software and in hardware.

It is important to note that if two symbols S_h and S_i agree, but S_i and S_j disagree, columns may be deleted in one or both of L_i and L_j. After this happens, it is possible for S_h to disagree with either of the modified symbols, and so S_h will have to be *re-agreed* with them. During that agreement, columns from L_h may have to be deleted, and so on. In this manner, a chain reaction of column deletions may occur. Hence, in order to ensure that a system of symbols gets to a pairwise-agreed

1. Produce a nonsingular transform matrix $U = U_{ij}$ of size $t \times t$ such that the product UA is a matrix with zeroes in its last $r = r_{ij}$ rows and of rank $t - r$. If $r = 0$, the symbols agree.
2. If $r > 0$, then compute the matrices UT_{ij} and UT_{ji}. Let Pr_{ij} denote the set of of UT_{ij}-column projections to the last r coordinates. If $Pr_{ij} = Pr_{ji}$, the symbols agree.
3. If $Pr_{ij} \neq Pr_{ji}$, first remove all columns from L_i whose UT_{ij}-associated column is such that its last-r-coordinate projection is not found in Pr_{ji}. Name the resulting matrix L'_i. Then similarly remove columns from L_j and name the resulting matrix L'_j. The symbols $A_i x = [L'_i]$ and $A_j x = [L'_j]$ agree.

Figure 2.1: Agreeing Two Symbols $A_i x = [L_i]$ and $A_j x = [L_j]$, where $L_\eta \in \mathbb{F}_2^{k_\eta \times s_\eta}$

state, we perform the *Agreeing1 Algorithm* in Figure 2.2 (see [32, Section 3.1]).

While the symbols in a system (2.1.2) do not pairwise agree,

1. find S_i and S_j which do not agree, and
2. agree S_i and S_j with the agreeing procedure in Figure 2.1.

Figure 2.2: Agreeing1 Algorithm

The time period in which the Agreeing1 Algorithm is run is called the *agreement phase*. Other approaches to agreeing are possible (and have been developed by the MRHS authors), but we do not focus on them. This is so that the software and hardware implementations discussed later are in line with each other to make comparisons between them easier to analyze. Indeed, one of these other approaches, the Agreeing2 Algorithm (see [32, Section 3.2]), is theoretically faster but requires much more memory, so for feasibility purposes we chose not to implement this approach.

Example

Suppose the variables X are $\{x_1, x_2, x_3, x_4, x_5\}$.

Let symbols $A_1X = [L_1]$ and $A_2X = [L_2]$ be given as

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

Algebraically, they correspond to the equations

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

We vertically join A_1 and A_2 to form A and row-reduce it, generating the matrix U:

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \to UA = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

Since the last two rows of UA are all-zero, $r = 2$. We construct T_{12} and T_{21}:

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

and compute UT_{12} and UT_{21}:

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

We examine the last r coordinates of columns in the above matrices and define

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

We note that the last r rows of the second and fourth columns of UT_{12} don't match any last r rows of columns of UT_{21}; that is, $(1,0) \notin Pr_{21}$. Hence, the second and fourth columns of L_1 should be removed. Also, the last r rows of the second and third columns of UT_{21} don't match any last r rows of columns of UT_{12}; that is, $(0,0) \notin Pr_{12}$. Hence, the second and third columns of L_2 should be removed.

The new symbols are

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

and they agree.

2.2.2 Gluing

After a system of symbols is in a pairwise-agreed state, we may choose to glue some symbols. The *gluing* of two symbols $S_i : A_i x = [L_i]$ and $S_j : A_j x = [L_j]$ is a new symbol $Bx = [L]$ whose set of solutions is the set of common solutions to $A_i x = [L_i]$ and $A_j x = [L_j]$. Once this new symbol is formed, it is inserted into the system and the two symbols S_i and S_j which formed it are no longer necessary and hence removed from the system.

Obtaining the matrix B is easy: with the notation in Figure 2.1, B is just the submatrix of UA in its last $t - r$ nonzero rows. The matrix L has $t - r$ rows and the columns are formed by adding one column from UT_{ij} to one column from

UT_{ji}. More specifically, we add a column from UT_{ij} and one from UT_{ji}, if they have the same projection to the last r coordinates. Reducing the sum to its first $t-r$ coordinates yields a column of L, and forming all such matching pairs yields the complete matrix L. Gluing two matrices L_i, L_j of width s_i and s_j may result in an L with as many as $s_i \cdot s_j$ columns. Consequently, we may not be able to afford to actually compute certain glues, and instead restrict to gluing only pairs of symbols where the number of columns in the resulting symbol does not exceed a certain *threshold*. The time period in which symbols are being glued is called the *glue phase*.

Once a pair of symbols has been glued, the resulting system will usually not be in a pairwise-agreed state, so the Agreeing1 Algorithm in Figure 2.2 can be run again, initiating another round of agreeing and gluing. The eventual goal of successive agreements and gluings is to obtain a system of symbols consisting only of a single symbol.

Example

The above symbols in the Agreeing example, when glued, will produce the symbol

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

2.2.3 Gluing Methods

The original MRHS specification [32] does not discuss which symbols to glue when it is time to do so, but such a choice can be important for software (and hardware) implementations. We discuss three methods below, but others are possible. None

of these methods will assign a symbol to be glued more than once in a given glue phase, and each method will, after a glue of two symbols, delete those symbols and keep the glue. Also, each has at its disposal a matrix of size expectations (calculated during agreement), so for any pair of symbols, it is known before the glue phase how many columns the L-part of the resulting glue would be.

First Available Method

In this method, the first symbol is examined to see if it can be glued to any other symbol. Once one is found, the assignment is made, and the next symbol is similarly examined.

Greedy Method

In this method, a list of tuples $(expectation, i, j)$ is sorted in increasing order, where i and j are symbol number indices, but all tuples where the expectation exceeds threshold are thrown out of the list. Then the list is examined and entries are chosen, starting with the leftmost entry (i.e., one with smallest expectation) provided that neither symbol index in the candidate entry appears in the previously chosen tuples.

Maximum Matching Method

In this method, a list of tuples is created as in the Greedy method, but then a graph is constructed such that the symbol indices are the vertices and the expectations are the edge weights. Then, a maximum matching algorithm is performed such that, first and foremost, the maximum possible number of matchings are constructed, and then of all such maximum matchings, the one with the total least sum of edge weights is chosen. An implementation of a related maximum matching algorithm can be found in [22].

2.2.4 Extracting Equations

From a given symbol $S: Ax = [L]$, where $L \in \mathbb{F}_2^{k \times s}$, we can try to extract *URHS (**U**nique **R**ight **H**and **S**ide) equations*: choosing an appropriate nonsingular transformation matrix V of size $k \times k$, the product VL is upper triangular with zeroes in its last r rows. Denoting by Pr the matrix formed by the VA-column projections to the last r coordinates, we obtain the r linear equations $Pr \cdot x = 0$. Next to these homogeneous equations, it may be possible to extract a nonhomogeneous linear equation: from the upper triangular matrix VL we can read off if the all-one-vector $(1\ 1\ \cdots\ 1)$ is in the span of the rows of L. If this is the case, we obtain the nonhomogeneous linear equation $(zA)x = 1$, where z is a row vector of length k such that $zL = (1\ 1\ \cdots\ 1)$.

Example

We can transform the symbol in the Gluing example above to

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

with obvious row transformations. The result implies three linear equations:

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

which are equivalent to the system of the two initial quadratic equations.

The time period in which equations are extracted from symbols is called the *equation extraction phase*. The resulting r or $r+1$ URHS equations can be combined and used in different ways, some of which involve the equation symbol discussed

below.

2.2.5 The Equation Symbol

An equation symbol (denoted S_0) can be used to help keep track of what the attack has learned about the equation system so far, and through the course of the attack, this symbol is constantly updated with new information and checked for inconsistency and size. Its A-part has the same number of columns as the A-parts of the other symbols, but its L-part has only one column. S_0 is not considered a proper part of the system (2.1.2), so it does not take part in the Agreement1 Algorithm, nor does it get destroyed after being glued to a symbol in the system. However, various implementations will agree and glue it to symbols at different times, and extracted equations will (in some form) wind up in the equation symbol. Further, information from guessing variables (described below) may also wind up in the equation symbol.

2.2.6 Guessing Variables

Owing to the chosen threshold, it may happen that a system is in a pairwise-agreed state, no new URHS equations can be computed, and no pair of symbols can be glued anymore. In such a situation, one is forced to guess a value of a variable. Before a guess is committed, the system of symbols—to which we will refer as the *state*—is stored. Then the guess is performed. Then, pairwise agreeing, computation of URHS equations, and gluing continue as normal. If after some steps the state, again, does not allow any new URHS equation to be extracted or pair of symbols to be glued, the state is again saved and another guess is committed.

Of course it is possible that in this process a guess for a variable is incorrect. This discovery manifests in the following manner: during the agreement of two

symbols, all right hand sides of at least one of the symbols get removed. When this happens, the state must be rolled back to a previous state, and a different guess must be made.

2.2.7 Guessing Methods

Depending on what the attacker wishes to accomplish, there are two dimensions of ways one can guess: Depth-First vs. Abbreviated vs. Clump, which addresses how incorrect guesses are handled, and Linear vs. Hot Bit, which addresses which key variables are guessed.

Depth-First Search

In a full MRHS attack (where no key bits are presumed to be known), the act of guessing a variable is done in a depth-first manner. When a key variable is to be guessed first, it is guessed as a 0, and another turn of agreeing (and possibly extraction) and gluing is performed. When it is time for another guess, another key variable is chosen, and its value is guessed as a 0. Then another turn, and so on until the key is found. If, after some guess the system is found to be inconsistent, the previous state is loaded and the last-guessed variable is guessed as a 1. If a key is not found and the system is inconsistent, the last-guessed variable is not guessed, the next-to-last-guessed variable is guessed as a 1, and then another turn is performed. This continues until the key is found (i.e., there is only one symbol remaining, or optionally, until the equation symbol has achieved maximum rank).

Abbreviated Search

Since the computation time in each turn is not trivial, it could be that a full attack may take an infeasible amount of time. To help get a handle on the time complexity

of a full attack, sometimes an abbreviated attack is performed, where the key bits are introduced one at a time. When a guess is first called for, a key variable is chosen, and its actual value is guessed, and another turn is performed. When another guess is called for, another variable is chosen, and its actual value is guessed, and so on until the whole key is found. Since the actual key values are used at each guess, the system will not devolve into an inconsistent state.

Clump Search

If the Abbreviated Search is not desired, one can perform a Clump Search, where a clump of key variables are chosen and guessed (as their actual values) at once when the first guess is called for. Subsequent guesses follow the Abbreviated Search approach. The number of variables guessed is a function of the number of symbols in the system after the first turn, as per Table 2.1.

symbols	bits to guess
513+	1
257-512	2
129-256	3
65-128	4
33-64	5
17-32	6
9-16	7
5-8	8
1-4	9

Table 2.1: Bits to Guess in Clump Search After First Turn

Since this method is exclusively used to help determine the performance of PET SNAKE (the reasoning for this will become clear once PET SNAKE is discussed), it is similar to Table 5.1. Additionally, this approach may have other applications; it might be showable that, for example, it is faster to guess a clump of variables at

a time instead of proceeding one-by-one, though this would be akin to a clumpish depth-first approach.

Linear Choice

In either the Depth-First, Abbreviated, or Clump approaches, one can guess key variables linearly, which is to say, start with the first key variable, and then the second, and then the third, and so on. (Alternately, one may reverse this choice, starting with the last key bit and work backwards.)

Hot Bit Choice

In this method, when a guess is called for, the equation symbol is examined to find the key variable which is involved in the most equations. If no such variable exists, the method reverts to making a linear choice until the next guess. It should be noted that this method of guessing is not part of the original algorithm; instead it has been developed by this author and independently by Schoonen [34].

2.2.8 Putting it All Together

There are a few different ways (called *flows*) one can implement an MRHS attack using the above components, and each has its own subtleties and performance characteristics. The goal of any flow is to agree, glue, guess, and (optionally) extract equations until only one symbol remains, or until the equation symbol (if employed) has maximum rank. If only one symbol remains, it will contain all solutions to the system at hand; in particular, if a plaintext/ciphertext pair was used, *all* keys that promote encryption from the given plaintext to the given ciphertext will be provided in the final symbol. We discuss a few such flows below, but certainly other flows are possible.

Basic Flow

In the Basic flow, we only agree, glue, and guess variables. No equations are extracted, and an equation symbol is not employed. In this approach, the system is agreed, glued, and guessed until only one symbol remains.

1. Enter the agreement phase: Each symbol is agreed to each other symbol until all symbols are pairwise-agreed.
   - If we produce a symbol whose L-matrix got all its columns deleted, then the system is inconsistent, so go to (3).
2. Enter the glue phase: If no glues are possible, save the state (that is, all the symbols) and then go to (4).
   - Pairwise glue symbols whose resultant's L-matrix has no more than the threshold number of columns.
   - If one symbol remains, terminate successfully. Otherwise, go to (1).
3. If a guess of a variable has not yet been made, terminate with failure as the original system has no solutions. Otherwise, roll back to a good state.
4. Make a new guess of the variables: Create a symbol corresponding to the guess and insert it into the symbol system. Go to (1).

Figure 2.3: Basic Flow

Schoonen Flow

The Schoonen flow [34, Algorithm 3.1] uses all components in a different particular way. Essentially, it loops through the actions of agreeing, extraction, and gluing, until the system doesn't change anymore, at which point it commits a guess and reenters the loop.

PET SNAKE Flow

The PET SNAKE flow is most closely related to the overall algorithm run in PET SNAKE (see Figure 4.2), and as such it is used to provide performance data to help

1. Enter the agreement phase: Each symbol is agreed to each other symbol until all symbols are pairwise-agreed.
    - If we produce a symbol whose L-matrix got all its columns deleted, then the system is inconsistent, so go to (7).
2. Enter the equation propagation phase: Equations are generated from each symbol, and then are row reduced, and then are row reduced against the current equation set, forming the new equation set.
    - If an inconsistency is found, go to (7).
    - If the new equation set is of maximum rank, terminate successfully.
    - If there is no new information in the new equation set, go to (5).
3. Make the new gather symbol from the new equation set and agree it to all symbols in the system.
    - If we get a symbol whose L-matrix got all its columns deleted, then the system is inconsistent, so go to (7).
4. Glue the gather symbol to all system symbols.
5. Enter the glue phase: Pairwise glue one pair of symbols whose resultant's L-matrix has no more than the threshold number of columns, if possible.
6. If a column in an L-matrix was deleted, or a new equation was extracted, or a pair of symbols was glued, go to (1).
7. If a guess of a variable has not yet been made, terminate with failure as the original system has no solutions. Otherwise, roll back to a good state.
8. Make a new guess of the variables: An equation corresponding to the guess is inserted into the equation symbol. Go to (1).

Figure 2.4: Schoonen Flow

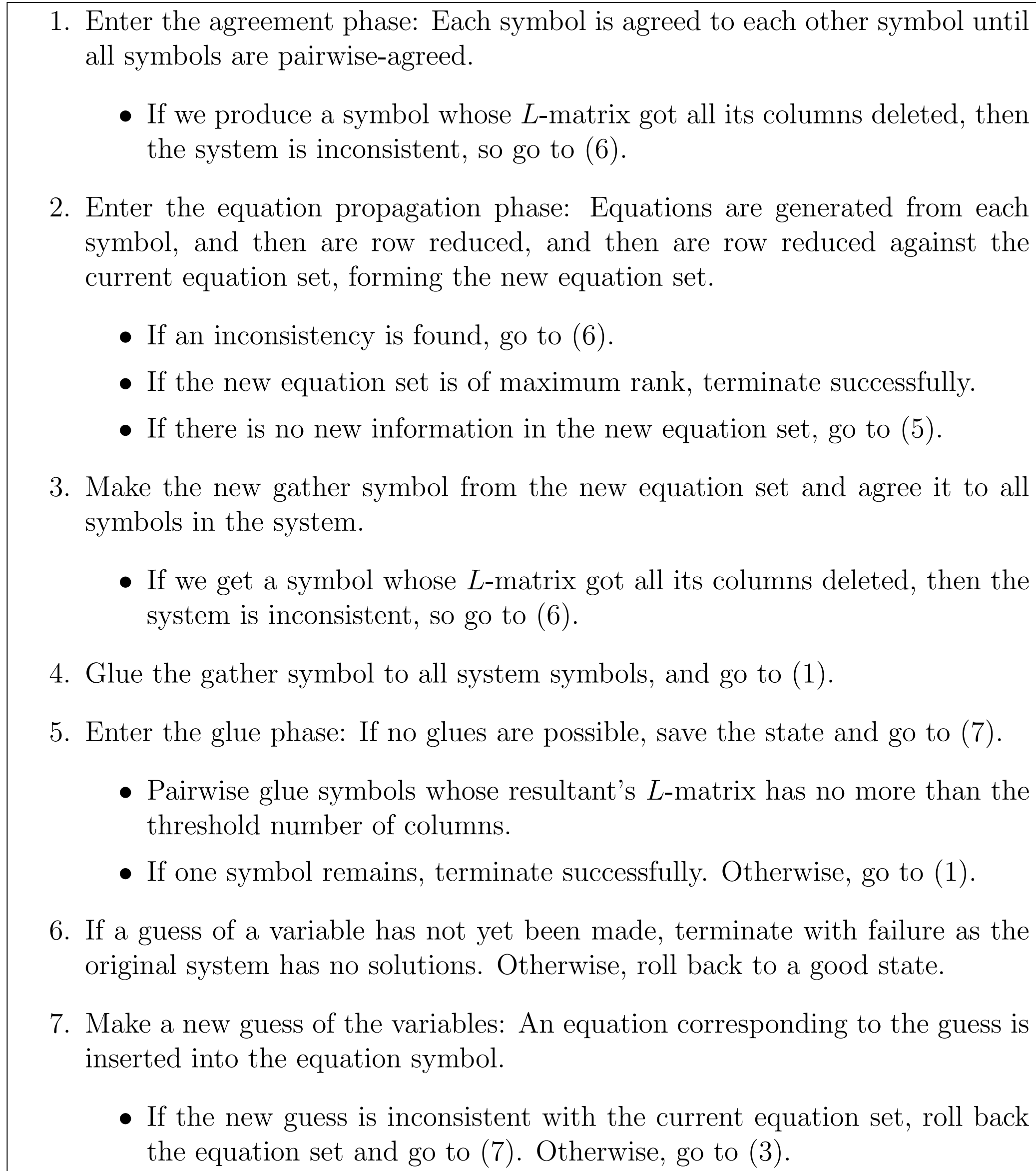
predict PET SNAKE's overall running time (see Chapter 5). This flow is similar to Schoonen's, except (a) upon finding new equations during extraction, pairwise agreement is performed again instead of attempting to glue them directly, and (b) it is not restricted to gluing just one pair of symbols.

Additional Terminology

A *segment* is a collection of alternating agreement and equation extraction phases that occur right before a glue phase. (If the chosen flow does not have equation extraction, a segment is just a single agreement phase.)

A *turn* is the collection of all the phases that occur before a guess is called for. For the PET SNAKE flow, a turn consists of a segment, followed by a glue phase, followed by a segment, followed by a glue phase, and so on, ending with a segment.

The *initial state* is the set of symbols that is constructed in the beginning before an MRHS attack is performed. The *initial turn* is the turn that is performed in the beginning on the initial state, that is, before any guesses are committed. Ideally an MRHS attack will only require the initial turn (i.e., no guesses), but often this is not the case. After the initial turn is complete, the system of symbols that remains is called the *base state*.



1. Enter the agreement phase: Each symbol is agreed to each other symbol until all symbols are pairwise-agreed.
   - If we produce a symbol whose L-matrix got all its columns deleted, then the system is inconsistent, so go to (6).
2. Enter the equation propagation phase: Equations are generated from each symbol, and then are row reduced, and then are row reduced against the current equation set, forming the new equation set.
   - If an inconsistency is found, go to (6).
   - If the new equation set is of maximum rank, terminate successfully.
   - If there is no new information in the new equation set, go to (5).
3. Make the new gather symbol from the new equation set and agree it to all symbols in the system.
   - If we get a symbol whose L-matrix got all its columns deleted, then the system is inconsistent, so go to (6).
4. Glue the gather symbol to all system symbols, and go to (1).
5. Enter the glue phase: If no glues are possible, save the state and go to (7).
   - Pairwise glue symbols whose resultant's L-matrix has no more than the threshold number of columns.
   - If one symbol remains, terminate successfully. Otherwise, go to (1).
6. If a guess of a variable has not yet been made, terminate with failure as the original system has no solutions. Otherwise, roll back to a good state.
7. Make a new guess of the variables: An equation corresponding to the guess is inserted into the equation symbol.
   - If the new guess is inconsistent with the current equation set, roll back the equation set and go to (7). Otherwise, go to (3).

Figure 2.5: PET SNAKE Flow

Chapter 3

Software

In order to discover the extent to which MRHS is effective, software implementations were constructed. The first such implementation of the three basic components of agreeing, gluing, and equation extraction was performed in MAGMA [8], but it seemed as though it consumed too much memory to be of practical use. This author then chose to implement everything by hand in C++ with the STL with a view to (a) optimize memory consumption, and (b) vary many aspects of an MRHS attack. The nine aspects that were varied are the flow choice, gluing method, guessing method, which cipher to attack, how many rounds of the chosen cipher to attack, the number of plaintext/ciphertext pairs the system can use, threshold, key choice, and whether or not to use optimizing hash tables. Metrics collection was also modularized and implemented so as to help calculate performance characteristics of each attack. The resulting code is referred to as the *codebase*.

We wish to stress that this chapter is not about optimizing individual software operations such as matrix multiplication and row reduction, for any implementor can make informed choices for these operations. Much has been done along these lines already [1], though we do take advantage of the Method of Four Russians for matrix multiplication.

3.1 An Explosion of Options

The flows, gluing methods, and guessing methods that were discussed in Chapter 2 were implemented with the exception of the Maximum Matching glue choice. The other options bear some discussion, below.

3.1.1 Cipher Choice and Rounds

Several ciphers were implemented and attacked using MRHS in different ways. We discuss each in turn.

AES

This cipher needs little introduction, though we attack its 128-bit variant specified in [30]. This variant uses 128-bit plaintexts and ciphertexts, with a 128-bit key. The complete cipher uses 10 rounds.

DESL

This block cipher is specified in [24], using 64-bit plaintexts and ciphertexts, with a 56-bit key. The complete cipher uses 16 rounds. It is extremely similar to DES, with the exceptions of each of the 8 different S-boxes in DES is replaced with one S-box in DESL, and the beginning and ending permutations are removed. Indeed, these two permutations add nothing to the complexity of DES. We chose to implement attacks on DESL because (a) little is known about DESL from a cryptanalytic perspective, and (b) results about MRHS attacks on DES are already published in [34], so results about DESL were deemed more valuable.

PRESENT

This block cipher is specified in [4], using 64-bit plaintexts and ciphertexts, with an 80-bit key. The complete cipher uses 31 rounds. The key schedule is nontrivial, as an S-box appears in it. Some attacks against PRESENT involve generating the whole code book, so it is natural to wonder if an MRHS attack can be effective against it.

KeeLoq

This cipher is currently used to secure communications between a key fob and the car it opens, using 32-bit plaintexts and ciphertexts, with a 64-bit key. The complete cipher uses 528 rounds, though much is already known about it cryptanalytically [11]. We felt it would be interesting to see how MRHS would stack up against this known data.

Settlers of KATAN

The KATAN cipher was recently introduced in CHES 2009 [10], using 32-bit plaintexts and ciphertexts, with an 80-bit key. The complete cipher uses 254 rounds, though little is known about it cryptanalytically. Hence, results about KATAN (or at least its 32-bit variant, which was the target of our attack) were deemed valuable.

A Sample Cipher from Stinson

In [41], Stinson proposes a modest block cipher for illustration purposes, using 16-bit plaintexts and ciphertexts, with a 32-bit key. The cipher uses 4 rounds to encrypt. Naturally, a brute-force attack is possible with this cipher, but we use it to explore if MRHS can be faster than a brute-force attack.

3.1.2 Multiple Plaintext/Ciphertext Pairs

An important innovation of this work is the introduction of the ability to use multiple plaintext/ciphertext pairs, as this has not been attempted before nor written about in the context of MRHS. Traditionally, an MRHS attack begins by the attacker creating an initial load of symbols corresponding to the use of only one PT/CT pair, and for all the ciphers listed above, this ability has been created in the codebase. However, we describe and implement two methods to use multiple pairs: the Multiple Symbol Method, and the Distributed Method.

Multiple Symbol Method

We observe that for each system, we can transpose the variables so that there exists an $e \in seg_y$ such that $x_1, x_2, \ldots, x_e$ correspond to all the key variables. If the system has a nontrivial key schedule, more key variables would be generated, so we permute and rearrange so that there exists $e' \in seg_y$ such that $e' > e$ and $x_1, x_2, \ldots, x_{e'}$ correspond to all the key variables in the entire key schedule. (If the system does not have a nontrivial key schedule, set e' to be e.)

Care is taken so that each system's initial load of symbols (constructed from one PT/CT pair) conforms to this structure, that the first e' variables are key variables, so the leftmost e' columns of each symbol's A-part address them.

If v PT/CT pairs are desired, v individual initial symbol sets are first generated, one for each pair. All of them have symbols whose A-parts have their first e' columns corresponding to the variables in the key schedule. Since e' is a function of the cryptosystem itself and not of the PT/CT pair chosen, e' is fixed across all of these v sets of symbols. Then the remaining $y - e'$ variables in each system correspond to interstitial variables in the actual encryption (called *state variables*),

not to variables in the key schedule.

So, we view a multiple PT/CT pair attack as an instance of a single PT/CT pair attack with e' key schedule variables and $p \times (y - e')$ state variables. We construct our initial load of symbols for our attack as follows: For each $i \in seg_v$, label the initial load of symbols generated from PT/CT pair i as P_i. Examine P_1 to find the symbols whose A-parts have 1s only in the first e' columns, and call this set of symbols K. (These would be the symbols corresponding to the key schedule. Note that the choice of P_1 is irrelevant since each P_i will have the exact same such symbols.) Let Q_i be $P_i - K$. For each $j \in seg_{\#Q_i}$, label the jth symbol's A-part as A_{ij} and the jth symbol's L-part as L_{ij}.

Now, for each $i \in seg_v$ and $j \in seg_{\#Q_i}$, let A'_{ij} be the result of augmenting A_{ij} by adding on the right a block of $v - 1$ groups of $y - e'$ columns of 0s, and then transposing the columns corresponding to the $y - e'$ state variables with the $(i-1)$st group of $y - e'$ columns just added. (For A_{1j}, the data stays put.)

In addition, for each $i \in seg_{\#K}$, let the ith symbol in K be called (B_i, M_i). Let B'_i be the result of augmenting B_i by adding on the right a block of $v - 1$ groups of $y - e'$ columns of 0s.

Then, the initial load of symbols for this attack is

$$\bigcup_{i=1}^{\#K} (B'_i, M_i) \ \cup \ \bigcup_{i=1}^{v} \bigcup_{j=1}^{\#Q_i} (A'_{ij}, L_{ij}).$$

Since the codebase has been optimized for memory consumption, some attacks are feasible with this many symbols.

Distributed Method

In the Distributed Method, many computers are used at once. Each computer is loaded with an initial symbol load corresponding to a different plaintext/ciphertext pair, though all encryptions are performed using the same key.

Then, each computer runs an MRHS attack on the set it has, and each computer is connected to each other computer via some message passing interface. At regular intervals, information about the equations in S_0 pertaining only to the first e' variables is transmitted, picked up, and rolled into each computer's equation extraction phases. If any computer detects an inconsistency or solution, all computers are notified and the process terminates.

For the actual implementation of this method, the MPICH2 implementation [23] of MPI was chosen for the message passing interface.

3.1.3 Other Options

Threshold

The threshold is often given as a number corresponding to the base 2 logarithm of the maximum number of columns the L-part of each symbol is allowed to be; for example, a threshold of 21 means that each L-part of each symbol was allowed to be as large as 2097152 columns, and if the system could glue two symbols but the resulting symbol would produce an L-part with more than 2097152 columns, that glue won't be performed.

Key Choice

In DES, if the parity bits of the key are ignored, the all-zero key was found to be weak [29], so a key based on this author's longtime friend and high school mathematics

partner, Katalina, was constructed by applying SHA-1 to her first name. The attacker can choose which key to encrypt against. Another use of this choice is discovering if the all-zero key would be weak in the other cryptosystems discussed. In the tables that follow, a "key choice of 0" refers to using the all-zero key, and a "key choice of 1" means a key based on the above hash.

Hash Table Usage

Often some additional memory is available during an agreement phase, so for each pair of agreed symbols, one can keep track of which partial columns are going to be added to which partial columns, should that pair of symbols be chosen to be glued. This naturally saves on time since the glue operation no longer has to recalculate these. If, however, the attacker wishes to use more memory for the state and associated calculations, this option can be turned off.

3.2 Software Attacks

Many attacks were performed, and data was collected for each attack. These *runs* are collected and analyzed in the following sections. Since the runs were performed on more than one computer, we describe the computers used in Table 3.1.

Name	Processor	Memory	Operating System
Pink	Dual-core Xeon X5260 3.33 GHz	48 GB	Win 7 Server Enterprise
Blue	Two quad-core Xeon E5520 2.26 GHz	24 GB	Win 7 Server Standard
Ritsuko	AMD Athlon 64 X2 3800+ 2.06 GHz	3 GB	Win XP Pro 64-bit
Yui	AMD Athlon 64 3500+ 2 GHz	3 GB	Win XP Pro 32-bit
Aoba	Pentium Dual-Core E2180 2 GHz	2 GB	Win XP Pro 64-bit
Ares	Pentium 4 2.8 GHz	1 GB	Win XP Pro 32-bit

Table 3.1: Computers Used in Execution

In addition, timing metrics were gathered. These are broken down into five quantities:

- AgrT, the total time spent in agreement phases
- GlueT, the total time spent in glue phases
- ExtrT, the total time spent extracting equations
- EagrT, the total time spent agreeing S_0 to the state
- EglueT, the total time spent gluing S_0 to the state

The sum of these is reflected in a quantity labelled TotalT.

Unless otherwise indicated, only one core of one processor was used to perform each attack, only one plaintext/ciphertext pair was employed, and all timings are in seconds.

3.2.1 Brute Force

It is natural to wonder if MRHS can be better than brute force. In at least one circumstance, the answer is 'yes'. The sample cipher from Stinson was attacked using MRHS on Ritsuko, with total runtime 15s, recovering all keys. Also, a brute force attack was carried out on the same machine, but encrypting with keys from 00000000 through 0007FFFF took 72 seconds. Hence, Ritsuko would take $2^{13} \cdot 72 = 589824$ seconds to go through the entire keyspace, so this MRHS attack has a factor 39321 improvement over brute force.

Additionally, four rounds of DESL were attacked on Yui with total runtime 38s, recovering all keys. A brute force attack was carried out on the same machine, but encrypting with keys from 00000000000000 to 0000000007FFFF took 189s. Hence,

Yui would take $2^{37} \cdot 189$ seconds to go through the entire keyspace, or about 823692 years.

Indeed, one gets the sense that if no guesses are required, an MRHS attack is much faster than brute force in general. We postpone the question of MRHS being better than brute force if guesses are required until Section 3.2.3.

3.2.2 Discovering δ

For more serious ciphers, very often the first turn of an MRHS attack will not be sufficient to discover the key, so guesses of the key variables must be committed. Naturally, the fewer guesses required, the better an attack is deemed to be. We give the name δ to the number of key bits we must guess before we discover the whole key through an MRHS attack. Many experiments were conducted in order to find and minimize this value.

Dependencies

It is important to note that δ is a function of the input parameters of an MRHS attack (less Hash Table Usage), but it is *not* a function of the particular machine implementing the attack. Execution times are a function of the particular machine, however, and it may happen that if the threshold is set too high, a lower-memory machine may use swap space rather heavily, rendering an attack infeasible. For any execution times provided, care was taken to ensure that swap space was not used.

Inconsistency Experiments

What is interesting is that if we guess the same δ variables in such a way that at least one guess is incorrect, often an inconsistency is detected, and further, if we guess less than δ of the same variables in any way, we still must guess at least one

more variable to determine inconsistency or the full key. This observation leads to a method of predicting running times. Since the larger δ is, the longer an attack takes, we are doubly motivated to find δ and ways of reducing it.

To verify this, an experiment was conducted whereby 4 rounds of PRESENT were attacked repeatedly on several Ares-type machines using a key choice of 1, PET SNAKE flow, threshold 20, Greedy Method Gluing, Abbreviated Linear Choice guessing, using optimizing hashtables. With these parameters, δ is 36. Each attack was similar to other MRHS attacks except for the following: one bit of the first 36 key bits was chosen uniformly at random and guessed incorrectly. The number of guessed bits required to determine inconsistency was recorded. This was performed 50 times. Then two bits of the first 36 were chosen uniformly at random to be guessed incorrectly, and the number of guessed bits required to determine inconsistency was recorded. This was performed 50 times. And so on, up to all 36 bits. The results are summarized in Table 3.2. So, for example, it was found that when 16 key bits were guessed incorrectly, on 11 of the 50 attacks it took the attack 39 key bit guesses to determine an inconsistency. In this experiment it is seen that, most of the time, an additional two or three bits are required to be guessed before an inconsistency is found. Nonetheless, this may be a function of the cipher itself, so we choose not to generalize this finding to all other ciphers, and proceed on our timing calculations in Chapter 5 as if we do not need to guess these two or three additional bits. This renders those calculations a bit optimistic, but nonetheless plausible, as (for example) the ‘36’ column here is often nontrivial.

No effort has been made to determine which specific bits (when guessed incorrectly) could be responsible for increasing the number of bits to guess, but it is expected that this characteristic is a function of the cipher itself. For particular ciphers it may be valuable to more thoroughly determine this.

Incorrect Key Bits Guessed	Required # of Bits to Guess						
	34	35	36	37	38	39	40
1	0	0	0	0	11	39	0
2	0	0	0	7	17	25	1
3	0	0	0	11	16	22	1
4	0	0	1	4	20	23	2
5	0	0	1	6	13	24	6
6	0	0	2	12	21	12	3
7	0	0	2	8	23	16	1
8	0	0	1	11	23	14	1
9	0	0	2	11	18	18	1
10	0	0	0	11	26	12	1
11	0	0	4	7	25	14	0
12	0	0	1	8	24	17	0
13	0	1	2	15	17	12	3
14	0	0	4	7	19	19	1
15	0	1	3	9	22	15	0
16	0	0	3	7	29	11	0
17	0	0	4	9	22	15	0
18	0	0	1	16	20	13	0
19	0	0	1	14	20	13	2
20	0	1	1	9	22	16	1
21	0	0	3	12	20	13	2
22	0	1	1	13	26	9	0
23	0	0	1	14	23	12	0
24	0	0	6	9	23	11	1
25	0	0	1	14	19	16	0
26	0	0	5	17	19	8	1
27	0	0	0	12	20	18	0
28	0	0	1	11	25	13	0
29	1	0	4	10	26	9	0
30	0	0	4	8	19	19	0
31	0	1	4	13	22	8	2
32	0	0	2	7	23	18	0
33	0	0	4	9	23	13	1
34	0	0	4	3	34	9	0
35	0	0	0	4	33	13	0
36	0	0	0	0	50	0	0

Table 3.2: Number of Attacks on 4 Rounds of PRESENT Requiring Listed Guessed Bits to Determine Inconsistency on Ares-type Machines

AES Varying Rounds

AES was attacked on Blue with varying rounds using fixed parameters: key choice of 1, PET SNAKE flow, threshold 20, Greedy Method gluing, Clump Search Linear Choice guessing, with the use of optimizing hashtables. The results are summarized in Table 3.3.

Rounds of AES	δ	AgrT	GlueT	ExtrT	EagrT	EglueT	TotalT
3	108	7413	107	6	317	602	8445
4	108	15123	257	29	440	826	16675
5	108	25867	560	56	536	1073	28092
6	108	40151	965	71	633	1347	43167
7	108	58343	1615	103	738	1626	62425
8	108	80376	2498	142	908	1825	85749
9	108	106382	3632	184	980	2118	113296
10	108	136844	5076	232	1163	2300	145615

Table 3.3: AES δ and Attack Runtimes on Blue, Varying Rounds

From this, we compute the timing of each function as a percentage of the total timing and display the results in Table 3.4.

Rounds of AES	AgrT %	GlueT %	ExtrT %	EagrT %	EglueT %
3	87.77	1.26	0.00	3.75	7.12
4	90.69	1.54	0.17	2.63	4.95
5	92.07	1.99	0.19	1.90	3.81
6	93.01	2.23	0.16	1.46	3.12
7	93.46	2.58	0.16	1.18	2.6
8	93.73	2.91	0.16	1.05	2.12
9	93.89	3.2	0.16	0.86	1.86
10	93.97	3.48	0.15	0.79	1.57

Table 3.4: AES Attack Runtimes on Blue as Percentages of Total Time

From this we see that the vast majority of the time of an MRHS attack is spent in the agreement phase. This trend does not waver through any of the other attempted attacks.

Further, we see that we have to guess 108 key variables before we get the entire 128-bit key. This is expected behavior, as it was found both in [32] and [34]. Indeed, this result follows their conjecture, which we restate for convenience:

Conjecture 3.1. *The δ for an MRHS attack employing Linear Choice guessing is equal to the key bits minus the (base 2 logarithm of the) threshold.*

Some other attacks follow this conjecture, as we shall see, but some do not.

Given that this δ is 108, it stands to reason that, using Depth-First Search, there would be something akin to 2^{108} turns to process just for the last guess, and since each turn takes nontrivial time, this would lead to an infeasible overall time cost. Hence, unless otherwise specified, any further attempts to find δ in software are done using an Abbreviated Search.

AES Varying Threshold

AES was further attacked on Blue as above, varying threshold, but with no optimizing hashtables. The results are summarized in Table 3.5.

	Rounds of AES							
Threshold	3	4	5	6	7	8	9	10
21	107	107	107	107	107	107	107	107
22	106	106	106	106	106	106	106	106
23	105	105	105	105	105	105	105	105

Table 3.5: AES δ on Blue Using Linear Choice, Varying Rounds and Threshold

Interestingly, these runs perfectly follow Conjecture 3.1.

DESL Varying Rounds and Threshold

DESL was attacked on Blue with varying rounds and threshold using fixed parameters: key choice of 1, PET SNAKE flow, Greedy Method gluing, Linear Choice guessing, using optimizing hashtables. The results are summarized in Table 3.6.

	Rounds of DESL						
Threshold	4	6	8	10	12	14	16
20	0	34	36	36	40	38	40
21	0	34	39	37	39	39	42
22	0	33	39	37	38	43	38
23	0	33	38	45	46	48	46

Table 3.6: DESL δ on Blue Using Linear Choice, Varying Rounds and Threshold

We can see from this data that four rounds of DESL can be handled in the initial turn of an MRHS attack, but things get more complicated with more rounds. Six rounds behave more or less in line with Conjecture 3.1, but for subsequent rounds it's not at all guaranteed that an increased threshold actually helps with the computation. Only for twelve rounds do we see an improvement with increased threshold, but once we move to a threshold of 23, δ increases dramatically. Hence, the conjecture fails, but it still may have value as a ballpark approximation of δ, should it be needed.

As an aside, we note that processing the full DESL with threshold 23 took about 5 days.

PRESENT Varying Rounds and Threshold

We notice that in both AES and DESL, the plaintext lengths are greater than or equal to the key lengths, so we would expect a very small number of keys to work for a given plaintext/ciphertext pair. However, PRESENT, KeeLoq, and KATAN are different in that their keylengths are larger than the plaintexts. Hence, we would expect multiple keys to work for the same plaintext/ciphertext pair.

PRESENT was attacked on Pink with varying rounds and threshold using fixed parameters: key choice of 1, PET SNAKE flow, Greedy Method gluing, Clump Search Linear Choice guessing, using optimizing hashtables. The results are summarized in Table 3.7. Blank entries correspond to attacks which were not performed.

	Rounds of PRESENT				
Threshold	4	6	8	12	16
21	36	56	61	60	59
22	35	56	60	58	61
23	36	55	60	59	

Table 3.7: PRESENT δ on Pink Using Linear Choice, Varying Rounds and Threshold

Further rounds are theoretically possible to attack on Pink, but were not performed owing to their expensive time cost. These δs exhibit a small departure from those conjectured. Given the δs found in Section 3.2.7, we expect similar results for later rounds.

KeeLoq Varying Rounds and Threshold

KeeLoq was attacked on Blue with varying rounds and threshold using fixed parameters: key choice of 1, PET SNAKE flow, Greedy Method gluing, Linear Choice guessing, without using optimizing hashtables. The results are summarized in Tables 3.8 and 3.9.

Threshold	Rounds of KeeLoq								
	64	96	128	160	192	224	256	288	320
20	12	17	44	45	49	49	51	49	51
21	11	15	45	44	51	48	50	50	51
22	10	16	43	42	50	49	49	48	49
23	9	15	44	43	49	49	47	48	48

Table 3.8: KeeLoq δ on Blue Using Linear Choice, Varying Rounds and Threshold

Threshold	Rounds of KeeLoq						
	352	384	416	448	480	512	528
20	54	52	54	55	53	52	54

Table 3.9: KeeLoq δ on Blue Using Linear Choice, Varying Rounds

From Table 3.8, we notice that sometimes an increased threshold will reduce δ, and sometimes not. Further, the δ values are a little higher than what Conjecture 3.1 would suggest. From Table 3.9, we note that δ is markedly higher (on average, 10 bits) than what is conjectured for those increased rounds.

KATAN Varying Rounds and Threshold

KATAN was attacked on Pink varying rounds and threshold using fixed parameters: key choice of 1, PET SNAKE flow, Greedy Method gluing, Clump Search guessing, using optimizing hashtables. The results are summarized in Tables 3.10 and 3.11. Blank entries correspond to attacks which were not performed.

	Rounds of KATAN						
Threshold	40	60	80	100	120	127	140
20	28	48	61	63			
21	27	46	58	67	69	69	69
23			59	70	75	63	70

Table 3.10: KATAN δ on Pink Using Clump Search, Small Rounds

	Rounds of KATAN					
Threshold	160	180	200	220	240	254
21	69	70	77	75	78	72
23	74	65	74	79	76	73

Table 3.11: KATAN δ on Pink Using Clump Search, Large Rounds

We notice that, from an MRHS perspective, KATAN is in no immediate danger of being broken. For 80 rounds we get approximately the δ that is conjectured, but for larger rounds, the δ required is substantially larger, at least 3 bits more and as much as 19 more.

We remark in passing that attacks of 60 rounds of KATAN produce 4 keys (including the used one), but attacks on 80+ rounds only produce one key.

3.2.3 Brute Force, Revisited

Guess Trees

To get a better picture of how a Depth-First Search MRHS attack would behave (which is to say, the attack one would use if there was no other knowledge of the key), we can imagine the MRHS process as a *guess tree* whose nodes symbolize points where the state was saved, and whose branches are the turns where a guess was committed. The root node of this tree symbolizes the initial state, from which only one branch follows downward corresponding to the initial turn. Then the base state is arrived at and stored. From the base state two branches hang, the left corresponding to the turn executed after 0 is guessed for the first key variable, and the right corresponding to the turn executed after 1 is guessed for that same variable. When future guesses are called for, the state is saved (corresponding to a node in the tree) and two more branches hang. During a turn, if a key is found or the state is determined to be inconsistent, a node is placed at the end of that turn's branch.

Using Guess Trees to Estimate Runtime

In an Abbreviated Search the times for all the operations per turn are recorded, so we can use this information to make an estimate for the time cost of a Depth-First Search. We assume that the time cost recorded for a turn committed after guessing a certain number of bits constitutes an average of the time cost we might see for each turn in the Depth-First Search guessing the same number of bits. Then, we just take the time cost recorded for the initial turn, add that to twice the time cost recorded for the turn in which one bit was guessed, add that to four times the time cost recorded for the turn in which two bits were guessed, and so on. The result

is our estimate for a Depth-First Search of an MRHS attack. Naturally, this will be a function of the particular machine used to carry out the Abbreviated Search attack.

Some Results

We give the results below with the caveat that the brute force key encryptions can stand to be optimized a bit, but nonetheless their time estimates can still serve as good signposts for when MRHS becomes relatively ineffective.

The brute force process for each machine and cipher uses keys starting from the all zero key, ending with 7FFFF prepended with the appropriate number of zeroes. We give the time this process took, and used it to give the estimate for how long it would take to search the entire keyspace. Finally, we give the expected MRHS Depth-First Search runtime calculated as discussed above.

We note that the values in Table 3.12 were calculated with Blue for AES, DESL, and KeeLoq, and with Pink for PRESENT and KATAN. All listed runs used a key choice of 1, PET SNAKE flow, Greedy Method gluing. The options presented in each entry correspond to the threshold used and the guessing method, where 'LI' is used to specify Linear Choice, and 'LH' is used to specify Hot Bit Choice; if neither of these two options are present, then the run used Clump Search.

We see that, in general, MRHS has the potential to be better than brute force right around when the corresponding δ is less than that predicted in Conjecture 3.1. Hence, MRHS seems to be favorable for small rounds of DESL, PRESENT, KeeLoq, and KATAN. Interestingly, MRHS does not seem to be favorable for any rounds of AES.

As an aside, we have a case where increased threshold helps the runtime of an MRHS Depth-First Search, but another case where increased threshold hurts it.

Cipher-Rounds and Options	7FFFF Time (s)	Brute Force Time (years)	MRHS DFS Time (years)
AES-10 t20	669	$13.768 \cdot 10^{27}$	$510.499 \cdot 10^{27}$
AES-9 t20	609	$12.533 \cdot 10^{27}$	$375.473 \cdot 10^{27}$
AES-8 t20	554	$11.401 \cdot 10^{27}$	$266.901 \cdot 10^{27}$
AES-7 t20	493	$10.146 \cdot 10^{27}$	$179.962 \cdot 10^{27}$
AES-6 t20	435	$8.952 \cdot 10^{27}$	$114.008 \cdot 10^{27}$
AES-5 t20	377	$7.758 \cdot 10^{27}$	$67.586 \cdot 10^{27}$
AES-4 t20	319	$6.565 \cdot 10^{27}$	$35.511 \cdot 10^{27}$
AES-3 t20	259	$5.330 \cdot 10^{27}$	$16.154 \cdot 10^{27}$
DESL-16 t20 LI	225	980 586	132 470 155
DESL-8 t20 LI	174	758 319	2 942 974
DESL-6 t20 LI	160	697 305	266 778
DESL-6 t20 LH	160	697 305	142 422
PRESENT-16 t21	255	$18.645 \cdot 10^{12}$	$806.689 \cdot 10^{12}$
PRESENT-8 t21	111	$8.116 \cdot 10^{12}$	$229.372 \cdot 10^{12}$
PRESENT-6 t21	98	$7.165 \cdot 10^{12}$	$3.427 \cdot 10^{12}$
PRESENT-4 t21	84	$6.141 \cdot 10^{12}$	558 836
KeeLoq-128 t20 LI	18	20 082 404	146 280 145
KeeLoq-112 t20	18	20 082 404	32 482
KATAN-127 t23	20	$1.462 \cdot 10^{12}$	$1847.005 \cdot 10^{12}$
KATAN-127 t21	20	$1.462 \cdot 10^{12}$	$73050.948 \cdot 10^{12}$
KATAN-80 t23	18	$1.316 \cdot 10^{12}$	$15.558 \cdot 10^{12}$
KATAN-80 t21	18	$1.316 \cdot 10^{12}$	$2.665 \cdot 10^{12}$
KATAN-70 t20	18	$1.316 \cdot 10^{12}$	$1.059 \cdot 10^{12}$
KATAN-60 t20	18	$1.316 \cdot 10^{12}$	196 379 628

Table 3.12: Estimated Brute Force and MRHS Depth-First Search Times

3.2.4 Flow Variations

To see the effect of using different flows, PRESENT was attacked using the Basic and PET SNAKE flows on Ritsuko with fixed parameters: Abbreviated Search, key choice of 1, threshold 20, Greedy Method gluing, using optimizing hashtables. The results are summarized in Table 3.13.

	Basic		PET SNAKE		
Rounds	δ	Time (s)	δ	Time (s)	Factor Increase
4	43	1687	36	2118	1.25
6	60	2960	57	10378	3.50
8	62	4473	61	21439	4.79
10	69	4432	65	44132	9.95
12	62	8895	62	60274	6.77
14	68	4749	66	101916	21.46
16	62	21458	61	109328	5.09

Table 3.13: PRESENT δ and Time on Ritsuko, Varying Rounds and Flow

We see that the use of equation extraction does in fact contribute to smaller δs, but at a nontrivial Abbreviated Search performance penalty. This doesn't immediately suggest that equation extraction in a Depth-First Search in software would be more costly than avoiding its use, however, since there are layers in the Basic software guess tree which are not present in that for equation extraction. (However, in the case of PRESENT, 12 rounds would stand to benefit from a Basic flow since the δs are the same.)

3.2.5 Guessing Variations

To see the effect of Hot Bit Choice, DESL was again attacked on Blue with varying rounds and threshold with fixed parameters: key choice of 1, PET SNAKE flow, Greedy Method gluing, using optimizing hashtables. The results are summarized in Table 3.14.

Threshold	Rounds of DESL				
	4	6	8	10	12
20	0	33	38	36	39
21	0	33	39	37	40
22	0	32	38	37	40
23	0	33	38	44	45

Table 3.14: DESL δ on Blue Using Hot Bit Choice, Varying Rounds and Threshold

This table is not too much different than Table 3.6, but most of the differences are where Hot Bit Choice takes one or two additional bits to guess. Six rounds seem to slightly favor Hot Bit Choice, however.

Additionally, four rounds of PRESENT were attacked on Pink with key choice of 1, PET SNAKE flow, Greedy Method gluing, using optimized hashtables. The results are summarized in Table 3.15.

Threshold	Linear Choice	Hot Bit Choice
20	36	39
21	35	40
22	35	40
23	36	33
24	35	34

Table 3.15: 4 Rounds of PRESENT δ on Pink, Varying Threshold and Guessing Method

So, for four rounds of PRESENT, Hot Bit Choice seems to be much worse at lower thresholds, and slightly better at higher thresholds. This is consistent with

the DESL results.

3.2.6 Glue Variations

To see if different gluing methods have an impact, PRESENT and DESL were attacked on Aoba using key choice 1, PET SNAKE flow, threshold 20, Clump Search, with optimizing hashtables. The results are summarized in Table 3.16.

Rounds of PRESENT	Greedy Method	First Available Method
4	36	39
6	57	56
8	61	61
10	65	60
12	62	60
Rounds of DESL		
8	36	38
12	40	41
16	40	38

Table 3.16: PRESENT and DESL δ on Aoba, Varying Rounds and Gluing Method

In both ciphers, larger rounds seem to favor the First Available Method of gluing over the Greedy Method, but there is an opposite preference in smaller rounds.

3.2.7 Key Variations

In the case of PRESENT, though one would not immediately expect the all-zero key to be weak, PRESENT was attacked on Yui (key choice 0) and Aoba (key choice 1) varying rounds with fixed parameters: PET SNAKE flow, threshold 20, Clump Search guessing, Greedy Method gluing, not using optimizing hashtables. The results are summarized in Table 3.17. Blank entries correspond to attacks which were not performed.

Rounds	Key Choice 0	Key Choice 1
3	32	29
4	35	36
5	45	45
6	58	57
8	61	61
10	64	65
12	65	62
14	65	66
16	60	61
18	62	61
20	61	63
22	61	63
24	61	62
26		62

Table 3.17: PRESENT δ on Yui and Aoba, Varying Rounds and Key Choice

From this table, we see that from an MRHS perspective, sometimes the all-zero key is a little weaker, and sometimes it is a little stronger, but we do not see an appreciable difference. This is not much of a surprise, since the all-zero key does not seem to be obviously weak from PRESENT's design.

We also notice that in both cases, past 8 rounds, the conjectured δ is 60, but the actual δ is a little higher.

3.2.8 Pair Variations

We define the *initial plaintext* as the leftmost appropriate number of bits from 0123456789ABCDEF0123456789ABCDEF, depending on the cipher to be attacked. Three ways have been devised to generate new plaintexts from old ones: the increment way, the rotation way, and the random way. The increment way just adds 1 to the rightmost bit, with carryover leftward. The rotation way rotates the plaintext left one bit with wraparound. The random way just constructs a new plaintext uniformly at random. Each way begins with the initial plaintext, and iterates for however many plaintexts are desired.

Distributed Method

Four rounds of PRESENT were attacked on 18 Ares-type machines with each way listed above and parameters keychoice 1, PET SNAKE flow, threshold 20, Greedy Method gluing, Linear Choice guessing, using optimizing hashtables. (Recall that δ for this attack with one plaintext/ciphertext pair is 36.) The results are summarized in Table 3.18.

Plaintext Way of Generation	δ
Increment	37
Rotation	35
Random	36

Table 3.18: 4 Rounds of PRESENT δ with Distributed Method on Ares-type Machines

Interestingly, we only make slight headway when we rotate our plaintexts, and incrementing was actually worse for this method. In general, though, it is a bit disappointing to only gain one bit for δ using 18 concurrent attacks, and this could just be a function of the cipher itself.

Additionally, six rounds of DESL were attacked on 17 Ares-type machines with each way listed above and the same parameters. (Recall that δ for this attack with one plaintext/ciphertext pair is 34.) It was found that in all three ways, δ was unchanged.

Multiple Symbol Method

Though the Multiple Symbol Method is very similar to the standard method one uses for one pair, the number of symbols in the initial load of p pairs is effectively p times the number of symbols in the initial load for one pair. Since each agreement phase agrees symbols pairwise, one can expect a time increase of a factor of $O(p^2)$ over the time required for an attack on one pair. Hence, this method is much more costly in time and memory than that used for a single pair, and so it comes as no surprise that fewer results were attainable.

AES. Ten rounds of AES were attacked on Pink using key choice of 1, PET SNAKE flow, Greedy Gluing, Abbreviated Linear Choice guessing, without using optimizing hashtables, but with using two pairs (Increment Way) instead of one. With threshold 21, δ is 107 (taking 7 days), and with threshold 22, δ is 106 (taking 10 days). For smaller rounds at threshold 21, additional results are summarized in Table 3.19.

Rounds	Two Pairs	Three Pairs
3	107	115
4	107	107
5	107	116

Table 3.19: AES δ on Pink, Varying Pairs

So, more pairs will be needed if we expect to see any improvement in δ at all.

Interestingly, δ increased in two cases with three pairs.

DESL. Sixteen rounds of DESL were attacked on Blue with key choice of 1, PET SNAKE flow, threshold 20, Greedy Method gluing, Clump Search Linear Choice guessing, without using optimizing hashtables, using up to five pairs (Increment Way). The results are summarized in Table 3.20.

Number of Pairs	δ	Time (hours)
2	40	32
3	38	99
4	38	276
5	40	745

Table 3.20: DESL δ and Time on Blue, Varying Pairs

Here, we actually see a contribution using three or four pairs, for δ is 40 with just one pair as per Table 3.6. However, two and five pairs yield no improvement. As attacking with five pairs took over a month, further pairs were not explored.

PRESENT. Four rounds of PRESENT were attacked on Yui (two and four pairs), Ritsuko (six pairs), and Aoba (eight pairs), all using Increment Way, key choice of 1, PET SNAKE flow, threshold 20, Greedy Method gluing, using optimizing hashtables. The results are summarized in Table 3.21.

Number of Pairs	δ
2	32
4	27
6	17
8	26

Table 3.21: 4 Rounds of PRESENT δ, Varying Pairs via Increment Way

This shows quite a dramatic improvement for PRESENT, but also with the puzzling δ of 26 for eight pairs.

A similar attack was performed on Ritsuko using the Rotation Way with results in Table 3.22.

Number of Pairs	δ
2	35
4	31
6	32

Table 3.22: 4 Rounds of PRESENT δ, Varying Pairs via Rotation Way

Here, the Rotation Way plaintext generation method is not as favorable, but it still yields improvements over one pair.

KeeLoq. KeeLoq was attacked on Ritsuko with key choice of 1, PET SNAKE flow, threshold 20, Greedy Method gluing, Clump Search Linear Choice guessing, without using optimizing hashtables, but using two pairs (Increment Way). 64 rounds yield a δ of 0 (finding 194 keys), a substantial improvement over the δ of 12 for one pair. 96 rounds yield a δ of 17 (finding one key), which is no improvement over one pair.

96 rounds were also attacked on Aoba, threshold 18, using four pairs (Increment Way), yielding a δ of 20, which also does not suggest an improvement.

KATAN. Forty rounds of KATAN were attacked on Ritsuko (threshold 20, no hashtables) and on Pink (threshold 21-23, with hashtables) using key choice of 1, PET SNAKE flow, Greedy Method gluing, Clump Search Linear Choice guessing, using two pairs (Increment Way). The results are summarized in Table 3.23.

The effect of varying threshold is once again mixed, with optimal results in all but the 21 case.

Additionally, 80 rounds were attacked with two pairs at threshold 20, giving a δ

Threshold	δ
20	0
21	9
22	0
23	0

Table 3.23: KATAN δ, Varying Threshold

of 63, and with three pairs at threshold 21, giving a δ of 62. These two results are slightly worse than for one pair.

Trends. It seems to be the case that multiple pairs do very little to help systems in which only one key was expected for a given pair (AES, DESL), but they can be helpful in system where multiple keys are expected for a given pair (PRESENT, KeeLoq, KATAN). Even so, this help may, or may not, be limited as the number of rounds increases in these systems; this suggests an avenue to explore for the future.

Chapter 4

Hardware

In this chapter, we present a special purpose architecture called PET SNAKE (**P**arallel **E**limination **T**echnique **S**upporting **N**ice **A**lgebraic **K**ey **E**limination) to implement MRHS in a dedicated hardware device. Its design is manufacturable within the current limits of fab technology. Further, it is designed to be scalable, and many of the components can benefit from future space efficiency gains in fab technology. As it happens, the limiting factor of such growth is the available bandwidth for interchip communication.

We discuss the ASIC design and cost in this chapter. Also, we compare the components of the design to the corresponding software components with a view to establish that they perform several orders of magnitude faster than software. In Chapter 5, we put those gains together to establish an overall improvement over software through an entire MRHS run.

Statement of Joint Work

The PET SNAKE design is co-authored with Dr. Rainer Steinwandt and Dr. Willi Geiselmann. Indeed, the design would be much weaker without either of their valuable contributions and insights.

Related work. A first (unpublished) proposal for using dedicated hardware to implement MRHS has been developed by Semaev in 2007, and, after modifications, recently been published in [35, 36]. The architecture described below has been developed independently and uses a very different approach. The use of special hardware for attacking a specific symmetric cipher has been proposed in [3]. In addition, numerous special purpose architectures for cryptanalytic purposes have been devised and discussed in the research literature—some prominent examples being TWINKLE [37, 25], TWIRL [38] and their successors [14, 17] for factoring integers, or Deep Crack [13] and COPACOBANA [19] for attacking DES. As linear algebra over $\mathbb{F}_2$ plays an essential role in MRHS, it comes to no surprise that our design benefits from available work related to the Number Field Sieve: For the row reduction over $\mathbb{F}_2$ we modify the linear algebra design SMITH of Bogdanov et al. [5, 6] to enable a more efficient handling of sparse matrices as occurring in the context of MRHS. (Note that SMITH has enjoyed previous success in [3].) The resulting JONES (**J**ustifiable **O**ptimization **N**eatly **E**nhancing **S**MITH) device might be of independent interest for other applications involving sparse matrices over $\mathbb{F}_2$.

The overall data flow in our architecture is remotely reminiscent of the systolic linear algebra design in [15], a main difference being the emphasis on a data flow of two dimensions. Two-dimensional data flows are well-known from special purpose designs for the Number Field Sieve, like [2, 26, 16], but the organization of the data flow in the new design is quite different and explains the choice of the acronym PET SNAKE for our architecture.

Structure of the chapter. Section 4.1 describes the overall architecture we use. The overall architecture uses several identical copies of a *main processing unit* whose

various components are explained in Section 4.2. Further details on the individual components are given in the appendix. Finally, Sections 4.3–4.5 analyze the expected performance of the complete device, comparing it with a software implementation of MRHS.

Implementation choices. The fundamental design parameters for PET SNAKE have been chosen in such a way that it is possible to host a complete system of symbols as needed for a key recovery attack on a modern block cipher like AES-128. For AES-128 specifically, the pertinent system of symbols involves 1,600 variables, and the initial system requires only 320 symbols. (For comparison, the initial system for PRESENT consists of more than 500 symbols, and also the number of variables is higher than for AES-128). In general, handling systems with no more than 2^{12} symbols still seems within the reach of PET SNAKE, and up to 2047 variables can be handled. For gluing symbols, we chose our threshold to be 2^{20} right hand sides. This seems a nice balance between the upper limits of software implementations and the upper limits of current hardware storage abilities. In light of the multiplicative nature of the growth of right hand sides during gluing, giving one or two more powers of 2 to the threshold does not seem to readily contribute to a significantly reduced running time.

In the described form, PET SNAKE has a storage capacity of 4.792 TB (not including the 'active' DRAMs in the traffic control chips), or enough to store 18,000 full-size symbols. This number was chosen based on the following observations: the symbol count for AES-128 will drop to 180 before threshold takes over, and it is possible we may have to guess up to 100 key variables before we find the key; hence up to 100 states may need to be stored along the way. Very rarely will a state actually be comprised of nothing but full-size (that is, 257 MB) symbols, so it will

almost always be possible to store more than 100 states; 400 or more states are not unlikely.

4.1 Overall Architecture

A complete PET SNAKE architecture consists of a several interconnected boards with each board hosting several ***M****ain* ***P****rocessing* ***U****nits* (MPUs). Each MPU is comprised of a small group of chips wired in a particular way, and there are $p \times p$ such MPUs placed in a grid across the individual boards, where $p = 2^{\lambda}$ is a power of 2. Subsequently we use $\lambda = 5$, yielding a total of $2^5 \cdot 2^5 = 1024$ MPUs, but the proposed architecture scales within reasonable limits; depending on the resources available, other parameter values, like $p^2 = 2^8$ might be an interesting option.

Each MPU can communicate with its north, south, east, and west neighbor MPUs (with no wraparound). For directing the action of the p^2 MPUs, a single ***M****aster* ***C****ontrol* ***P****rocessor* (MCP) is used. The MCP will make most of the decisions regarding which symbols to send where, which symbols to glue, and when to guess a variable.[1] The MCP, which sits in a north corner, has *agents* which sit at the north end of the board, one per column. Each agent has a southbound bus that connects to each MPU in that column via 'hops' between MPUs, so each off-chip part of the bus is short. Each agent communicates to the MCP horizontally via 'hops' between agents. Figure 4.1 gives a schematic view of the overall architecture.

[1]By replacing the MCP, the overall algorithm can be changed, e. g., to accomodate a different MRHS variant.

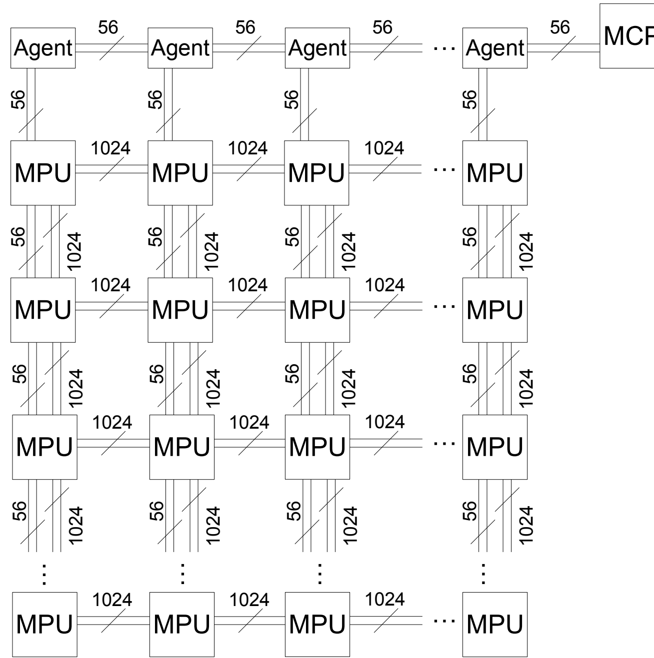


Figure 4.1: Overall architecture of PET SNAKE

4.1.1 Initialization

The initial system of n symbols is derived from the cipher, and a solution to the system of symbols yields a secret key for the attacked symmetric cipher that is consistent with the particular plaintext/ciphertext pair(s) used. The symbols are loaded onto the p^2 MPUs as evenly as possible. Let g be the number of symbols stored in each MPU—should the symbol count not be evenly divisible by the number of MPUs, we imagine empty symbols to fill in the gaps. Now imagine labelling each symbol in each MPU with a number in $\{1, \ldots, g\}$. We call all symbols labelled with the same number a *snake*. Hence we have g snakes. If at this point $g = 1$, we halve the number of MPUs to use, redistribute the symbols to this half, and try again; we continue this process until $g = 2$. The collection of MPUs now occupied with symbols is called the *active area* for this computation. Any inactive MPUs will be taken advantage of with parallelism, discussed later. The MCP determines

a Hamiltonian cycle through all MPUs, i. e., a path through all the MPUs such that one can move from one MPU to one of its neighbors in a closed circuit, without visiting the same MPU twice. The MCP will do the same for smaller groups of MPUs: $p \times \frac{p}{2}$, $\frac{p}{2} \times \frac{p}{2}$, $\frac{p}{2} \times \frac{p}{4}$, etc.—all the way down to 2×1. This data can be hardwired into the MCP, and we may assume that the MCP knows a path for each possible size active area.

4.1.2 Processing of Symbols

During a computation, it may happen that the symbol count n drops below the number of MPUs used to process them. If this happens, we move the symbols so that only half of the MPUs will be occupied with symbols. (This guarantees $g = 2$.) The active area is then halved. Any inactive MPUs will be taken advantage of with parallelism. Hence, at all points in the process, if g is not a power of 2, it will proceed as if g were the next highest power of 2 for board divisibility purposes. The overall algorithm run by PET SNAKE is summarized in Figure 4.2.

Before going into details of the overall algorithm, we mention that, to the best of our knowledge, the existing theoretical analysis of MRHS does not allow a precise prediction of how often the individual steps in Figure 4.2 are to be performed. This problem is not specific to PET SNAKE and arises for software implementations as well. For the subsequent analysis this means that we focus on judging PET SNAKE's performance *relative* to a software implementation. *Absolute* running times obviously depend on the particular block cipher/system of symbols, and we develop some theoretical tools to predict these in Chapter 5.

1. Enter the agreement phase: Each symbol is agreed to each other symbol until all symbols are pairwise-agreed.
   - If we produce a symbol whose L-matrix got all its columns deleted, then the system is inconsistent, so go to (6).
2. Enter the equation propagation phase: Equations are generated from each symbol, and then are row reduced, and then are row reduced against the current equation set, forming the new equation set.
   - If an inconsistency is found, go to (6).
   - If the new equation set is of maximum rank, we terminate successfully since a key has been found.
   - If there is no new information in the new equation set, go to (5).
3. Make the new gather symbol from the new equation set and agree it to all symbols in the system.
   - If we get a symbol whose L-matrix got all its columns deleted, then the system is inconsistent, so go to (6).
4. Glue the gather symbol to all system symbols, and go back to (1).
5. Enter the glue phase: If no two symbols can be glued such that the result's L-matrix has no more than 2^{20} columns, save the state (that is, all the symbols and the equation set) in the MPUs and then go to (7).
   - If necessary, move symbols so that any given pair of symbols to be glued appear in the same MPU.
   - Pairwise glue the symbols whose resultant's L-matrix has no more than 2^{20} columns. Delete the symbols which contributed to each glue.
   - If necessary, move symbols among the MPUs so that they have the same number of symbols. If there are less symbols than MPUs, move the symbols so they only occupy half the MPUs. This halves the active area.
   - If one symbol remains, terminate successfully as keys have been found. Otherwise, go to (1).
6. If a guess of a variable has not yet been made, terminate with failure as the original system has no solutions. Otherwise, roll back the symbols to a good state.
7. Make a new guess of the variables: The head MPU loads the equation set into its row reducer and introduces a row corresponding to the guess.
   - If the new guess is inconsistent with the current equation set, roll back the equation set and go to (7). Otherwise, go to (3).

Figure 4.2: Overall algorithm run by PET SNAKE

4.1.3 PET SNAKE's Agreement Phase

The majority of activity on the board will be during the agreement phase. This is broken down into k stages, where $k = \lceil \log_g n \rceil$.

First Stage

In the first stage, the entire active area is used. All but one snake (i. e., snakes 1 through $g - 1$) stay put on the MPUs. On each MPU, the symbol in the *motile snake* (i. e., snake g) is agreed to every other symbol on that MPU. When the last such agreement is taking place, the MPU sends the motile snake's updated symbol (that is, with deletions incorporated) to the next MPU in the active area's path. Since this is happening simultaneously for all MPUs in the active area, each MPU gets the next symbol in the motile snake. This continues q times, where q is the number of MPUs in the active area. If a deletion has occurred somewhere in this process, the MCP records the affected symbol's number, but otherwise continues normally.

Now, snake g will be fixed, and snake $g-1$ will move. The only difference here is that symbols from snake $g-1$ will not need to be agreed with those from snake g since that agreement has already been performed. After q times, snakes g and $g-1$ will be fixed, but snake $g-2$ will move. And so on. If a deletion has occurred for any of the g snakes, the MCP moves the affected symbols into larger-numbered snakes (e. g., g, $g-1$) and moves unaffected symbols into smaller-numbered snakes. Often this is just a renumbering inside an MPU, so no movement happens in these cases. Then the first stage is repeated again, noting that if all the symbols in a lower-numbered snake have no deletions in the previous run, it is not required to become motile. If a deletion occurs, the MCP repeats the process of moving affected

symbols and starting the stage again.

Second Stage

At this point, all snakes are agreed to all other snakes, but the symbols within each snake still need to be addressed. The active area is split up into g *stage areas*, each with q/g MPUs. For each $1 \leq j \leq g$, symbols from snake j move to stage area j. After this move is complete, we relabel each symbol in each MPU so that different snakes are formed, but the snakes only move in their given stage area. Hence, each snake is $1/g$ the size it used to be. Now, the same process is performed as in the first stage, but with smaller snakes and smaller paths.

If a deletion has been recorded in this stage, the stage is allowed to complete, but not recur nor go into the next stage. Then the affected symbols (from *all* stage areas) are grouped together into one (or possibly more) q-sized snakes with large snake numbers, they are moved into appropriate positions, and the first stage is entered again.

Subsequent Stages

If the second stage records no deletions, we continue this process of dividing the snakes and the stage areas by g until the stage area is one MPU. (Deletions found in any subsequent stage are handled the same way as described in the second stage.) At the last stage, the g symbols comprise g snakes of size 1 each, and so they are simply agreed to each other inside that MPU.

Time Estimate

The initial load's symbols will most likely have A-parts whose 1s are in different positions, so any particular pair of symbols will likely be already agreed, so no

deletions will occur. After the first glue, it is still likely no deletions will occur. After the second glue, however, things get less predictable, but by this point the symbol count will drop by a factor of 4. (In the case of AES-128, the threshold will take hold before the second glue, so we can only expect the symbol count to halve before guesses must be performed.) After these initial turns, deletion prediction becomes much less obvious, and it is certainly possible to go through many agreement phases before considering a glue. Handling deletions is needed in both software and hardware implementations, and it seems fair to consider PET SNAKE's efficiency in handling deletions as being at least comparable to that of a software implementation (see Section 4.4.2 and Appendix A). To get a handle on a time estimate for PET SNAKE's agreement phase we consider only the case that no deletions will occur. (We will see how these computations become helpful in the general case in Chapter 5.)

We note that per stage there are $g(g-1)/2$ agreements per MPU, and this happens q times in the first stage, q/g in the second, and so forth, up to 1 in the last. Since $g = n/q$, adding up the costs we have

$$\sum_{i=0}^{k-1} \frac{g(g-1)}{2} \cdot \frac{q}{g^i} = \frac{g(g-1)}{2} \cdot \frac{n}{g} \cdot \left(\frac{1 - \frac{1}{g^k}}{1 - \frac{1}{g}} \right) = \frac{(g-1)n}{2} \cdot \left(\frac{\frac{n-1}{n}}{\frac{g-1}{g}} \right) = \frac{g(n-1)}{2}$$

total agreements. Since we try to arrange things so that g is 2 as often as possible, this translates into $n-1$ agreements in these cases.

What is not included so far is the time of moving symbols between stages. Let the active area have dimensions $q_1 \times q_2 = q$ where $q_1 \leq q_2$, and suppose g is 2. After the first stage, a symbol moves along the longer dimension, but halfway so that it can find its new position. Another symbol from that position must get to where the first started, so they both must use those directions. This will introduce a factor

two slowdown in all movement calculations. Hence, after the first stage it takes $2 \cdot \left(\frac{q_2}{2}\right)$ moves to get the symbols into their new positions, and the stage area then has dimensions $q_1 \times \frac{q_2}{2}$. We alternate which dimension we travel on in each stage, so the next stage cost is $2\left(\frac{q_1}{2}\right)$. Then $2 \cdot \left(\frac{q_2}{4}\right)$, then $2 \cdot \left(\frac{q_1}{4}\right)$, and so on. Presuming k is even, this gives a time estimate of

$$(q_1 + q_2) \cdot \sum_{i=0}^{\frac{k}{2}-1} \frac{1}{2^i} = (q_1 + q_2) \cdot \left(\frac{1 - \left(\frac{1}{2}\right)^{k/2}}{1 - \frac{1}{2}} \right) = 2 \cdot (q_1 + q_2) \cdot \frac{2^{k/2} - 1}{2^{k/2}} < 2 \cdot (q_1 + q_2)$$

total moves for the whole agreement phase.

The situation for $g = 4$ is not as easy, since symbols have to move to different quadrants of the active area $q_1 \times q_2$. We observe that it must be the case that $q_1 = q_2$, since the only time we might have $g > 2$ is in the beginning, when we have the full board at our disposal.

Hence, we perform a sort of rotation, where each quadrant of symbols (one symbol per MPU per move) moves to the next clockwise (or counterclockwise) quadrant simultaneously. This is possible since all four directional buses of each MPU can be used simultaneously, and no directional bus needs to be used more than once at a time. After the first stage, in the first rotation the symbols whose target locations are in the diagonal quadrant move $\frac{q_1}{2}$ in one direction. In the second rotation, these same symbols move $\frac{q_1}{2}$ in the appropriate perpendicular direction to get to their target location. In the third rotation, symbols whose target quadrant are clockwise of them will move $\frac{q_1}{2}$ in that direction. The fourth rotation is similar to the third, but for counterclockwise-bound symbols. Thus, we have $4 \cdot \left(\frac{q_1}{2}\right) = 2 \cdot q_1$ moves for this stage. Subsequent stages are similar but the distance is half of the

previous distance. Thus we have

$$\sum_{i=0}^{k-1} 2 \cdot \left(\frac{q_1}{2^i}\right) = 2 \cdot q_1 \cdot 2 \cdot \left(\frac{2^k - 1}{2^k}\right) < 4q_1$$

total moves for the whole agreement phase.

4.1.4 PET SNAKE's Equation Propagation Phase

During agreement, it is recorded whether a symbol had columns deleted. PET SNAKE will extract equations from such symbols using each MPU simultaneously and gather them all (together with the current equation set) into a *gather symbol*, which is then agreed and glued to every symbol. The propagation phase consists of either one or two extraction stages (depending on if g is 2 or 4, respectively) followed by the resolution stage, followed by the propagation stage.

Extraction Stages

In the first extraction stage, equations from all symbols in snake 1 are extracted simultaneously and stored in each MPU. Then equations from all symbols in snake 2 are extracted simultaneously. All equations that have been extracted are then *mass row reduced* down to at most 2047 equations. To illustrate this process, first, imagine a label number from 0 through $q - 1$ for each MPU in the path. (Label 0 is given to the *head* MPU, which sits in the upper left corner of its active area. Label 1 is given to the next MPU in the Hamiltonian cycle. And so on. For ease of discussion, we also define the notation $x \equiv_m y$ to mean that m divides $x - y$, or alternately, x is congruent to y modulo m.)

Mass row reduction is then accomplished by the following process: each MPU row reduces the equations from its symbols in snakes 1 and 2. Then the MPUs with

labels $\equiv_2 1$ send their results to the MPU with label 1 less. Now those MPUs with labels $\equiv_2 0$ have up to 4094 equations, and each row reduces its set. This results in no more than 2047 equations. Then the MPUs with labels $\equiv_4 2$ send their resulting equations to the MPU with label 2 less. Another row reduction takes place. Then the MPUs with labels $\equiv_8 4$ send their resulting equations to the MPU with label 4 less. And so on, until equations get to the head MPU and are row reduced. These results are then stored.

If there is a second extraction stage, equations from symbols in snakes 3 and 4 are extracted and mass row reduced to at most 2047 more equations (which will also lie in the head MPU); these are then row reduced with the previous group of equations. The result is a group of at most 2047 equations called the *gather equations*.

Resolution Stage

The head MPU will then retrieve from storage the current *equation set*—which corresponds to the symbol S_0 in [34, Section 3]. (In the beginning, the equation set consists of no equations.) Then this is row reduced with the gather equations and the result is checked for consistency. If an inconsistency is found, this is signaled to the MCP; the MCP will then deem the current guess incorrect and move on to a new guess. If no inconsistency is found, the result is checked for maximal rank (i. e. number of nontrivial rows equal to n). If it has maximal rank, the MCP is alerted that a solution has been found. Otherwise, the result is stored as the new equation set. This is checked to see if there is a new equation that was not in the old equation set via a row count. If there is no new information, the glue phase begins; else, the propagation stage begins.

Propagation Stage

The head MPU creates the gather symbol and sends it to its east neighbor, and after that is done, it sends it to its south neighbor. The east neighbor will store it and then send it to its east neighbor, and then its south neighbor. And so on for all MPUs in the top row of the active area. An MPU that received the symbol from its north neighbor merely stores it and sends it to its south neighbor. Once all MPUs receive the gather symbol, it is agreed to every symbol in the MPU, with the results of the agreements propagated to the next MPU in the Hamiltonian cycle. As with normal agreement, if every column of a symbol's L-part gets deleted, the MPU signals the MCP that an inconsistency is found. Otherwise, after all agreements are complete, each MPU glues the gather symbol to each symbol it has.

Time Estimate

Since there are g symbols in an MPU and each MPU extracts simultaneously, we pay the time cost of an extraction g times. There are $\frac{g}{2}$ mass row reductions, each comprising $\log_2 q + 1$ row reductions and $1 + 2 + 4 + \cdots + \frac{q}{2} = q - 1$ moves of at most 2047 equations. (Moving one such equation set is much faster than moving a symbol, since an equation is expressed in 2048 bits.) In the case of two extraction stages, we row reduce an additional time. Propagating the gather symbol takes $q_1 + q_2$ moves, and finally since each MPU agrees, and then glues, simultaneously, we pay the agreement time of two symbols g times and the glue time g times.

4.1.5 PET SNAKE's Glue Phase

Since the MCP knows which pairs of symbols will glue to produce a symbol with 2^{20} or less columns, it merely directs moves to get these pairs into MPUs, and then

the MPUs glue them in parallel. The number of moves needed is not completely predictable, but we observe the following: in the early stages of the algorithm, a given symbol can be glued to almost every other symbol, so in particular each MPU won't have to move any symbols at all before gluing. In the later stages of the algorithm, very few glues are called for (often only one or two), so symbols can be moved directly to where they need to go. Since the active area is $q_1 \times q_2$ MPUs, this constitutes at most $q_1 + q_2 - 2$ moves.

Whatever the case, we can always elect to move symbols in the following manner: for each pair of symbols to be glued, label one member as a first component and the other as a second component. Symbols that are not to be glued remain unlabelled. If $g = 2$ and there are two first components in an MPU, relabel one as a second component and relabel its mate as a first. Do this again if the new labelling causes another double. And so on. Note this process cannot result in an infinite loop. Perform a similar process for MPUs with two second components. If $g = 4$ and there are three or more first components (or three or more second components) in an MPU, perform a similar relabelling process.

Now, we move symbols along the snake in a two-stroke process. In the first stroke, we move an out-of-place second component (or failing that, an unlabelled symbol) from MPU 0 to MPU 1, from MPU 2 to MPU 3, and so forth. In the second stroke, we move an out-of-place second component (or failing that, an unlabelled symbol) from MPU 1 to MPU 2, from MPU 3 to MPU 4, and so forth. Observe that an MPU keeps a second component if it also has the associated first component. This results in $q - 1$ moves if $g = 2$, or $2(q - 1)$ moves if $g = 4$.

The glue time, however, is in general higher than an agreement time. With $g = 2$, we only pay the glue time once, since each MPU will be gluing all gluable pairs in parallel with none waiting to be glued. With $g = 4$, we pay the glue time

at most twice; in general, the glue time is paid at most $g/2$ times.

4.1.6 Parallelism

Once the active area becomes half the original board (or less), and a guess is required, the MCP considers performing a parallel computation on the inactive area. The MCP will make a guess for a key variable in one area, and make the opposite guess for the same key variable in the other. Then both areas will be considered active areas, but their computations will be completely separated.

4.2 Main Processing Unit

The MPU is a collection of seven chips comprising five functional units, each with its own responsibilities and behavior. We discuss each functional unit in turn: the traffic controller, the row reducer, the multiplier, the hash table, and the adder. Each functional unit is connected to a 2048-bit-wide bus called the *MPU bus*.

4.2.1 MPU Data Flow

We describe the sequence of events that will occur inside each MPU when it is agreeing, when it is extracting equations, and when it is gluing. The particular details of each component are discussed in the sections below. Figure 4.3 gives an overview of how most of the components are interconnected. (The traffic controller sits on the north end of the MPU bus, directing traffic between it and other traffic controllers of other MPUs.)

The high level order of operations during an agreement between two symbols S_i and S_j is as given in Figure 4.4, and the (somewhat similar) procedure for gluing two symbols S_i and S_j is described in Figure 4.5. Finally, Figures 4.6 and 4.7 list

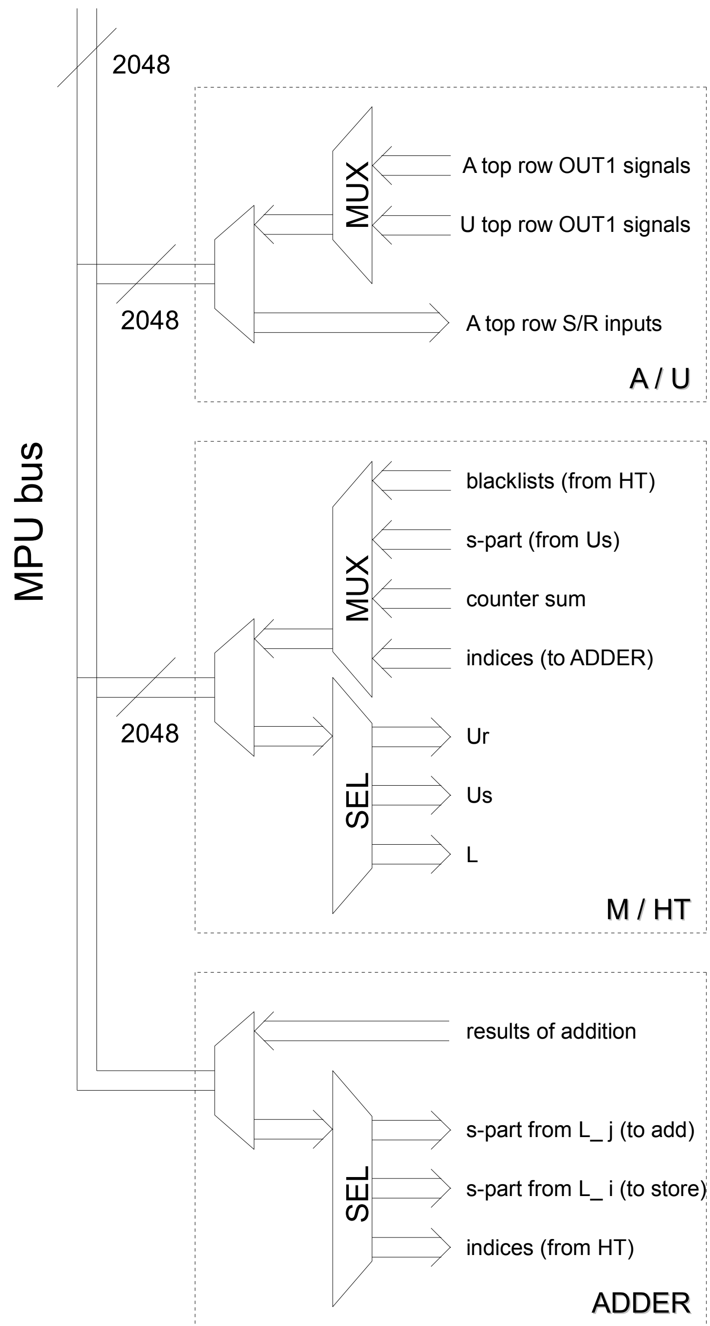


Figure 4.3: MPU Busing Diagram (High Level)

the high level order of operations for extracting equations from a symbol and for a mass row reduction, respectively.

1. A_i is sent across the MPU bus and the row reducer picks it up.
2. A_j is sent across the MPU bus and the row reducer picks it up.
3. The row reducer calculates both B and U.
4. The row reducer determines if r is 0. If $r = 0$, terminate with agreement signal. Otherwise,
5. The row reducer sends the left cols(L_i) part of U across the MPU bus to the multiplier.
6. For each column c of L_i:
   - c is sent across the MPU bus and the multiplier picks it up.
   - The multiplier sends its r-part to the hash table.
   - The hash table stores an indicator that that r-part has been created.
7. The row reducer sends the right cols(L_j) part of U across the MPU bus to the multiplier.
8. For each column d of L_j:
   - d is sent across the MPU bus and the multiplier picks it up.
   - The multiplier sends its r-part to the hash table.
   - If the r-part had been formed by L_i, the hash table stores an indicator for this.
   - If not, the hash table reports the column index of d across the MPU bus to be deleted.
9. For each entry in the hash table's buffer DRAM, if the entry is not found in the table itself, the column index is reported across the MPU bus to be deleted.
10. If no deletions have been recorded, the hash table sends the value of its glue counter across the MPU bus to the traffic controller.

Figure 4.4: High Level Order of Operations During an Agreement

Subsequently we discuss the individual components of an MPU, but for the sake of readability postpone low-level details and area estimates to the appendix.

1. A_i is sent across the MPU bus and the row reducer picks it up.

2. A_j is sent across the MPU bus and the row reducer picks it up.

3. The row reducer calculates both B and U, determines if r is 0, and sends B across the MPU bus to be stored.

4. The row reducer sends the left cols(L_i) part of U across the MPU bus to the multiplier.

5. For each column c of L_i:

   - c is sent across the MPU bus and the multiplier picks it up.
   - The multiplier sends its s-part to the adder for storage.
   - If $r \neq 0$, the multiplier sends its r-part to the hash table, and the hash table stores the L_i column index that gave rise to the r-part.

6. The hash table re-examines its DRAM buffer, possibly sending pairs of data across the MPU bus to the adder.

7. The row reducer sends the right cols(L_j) part of U across the MPU bus to the multiplier.

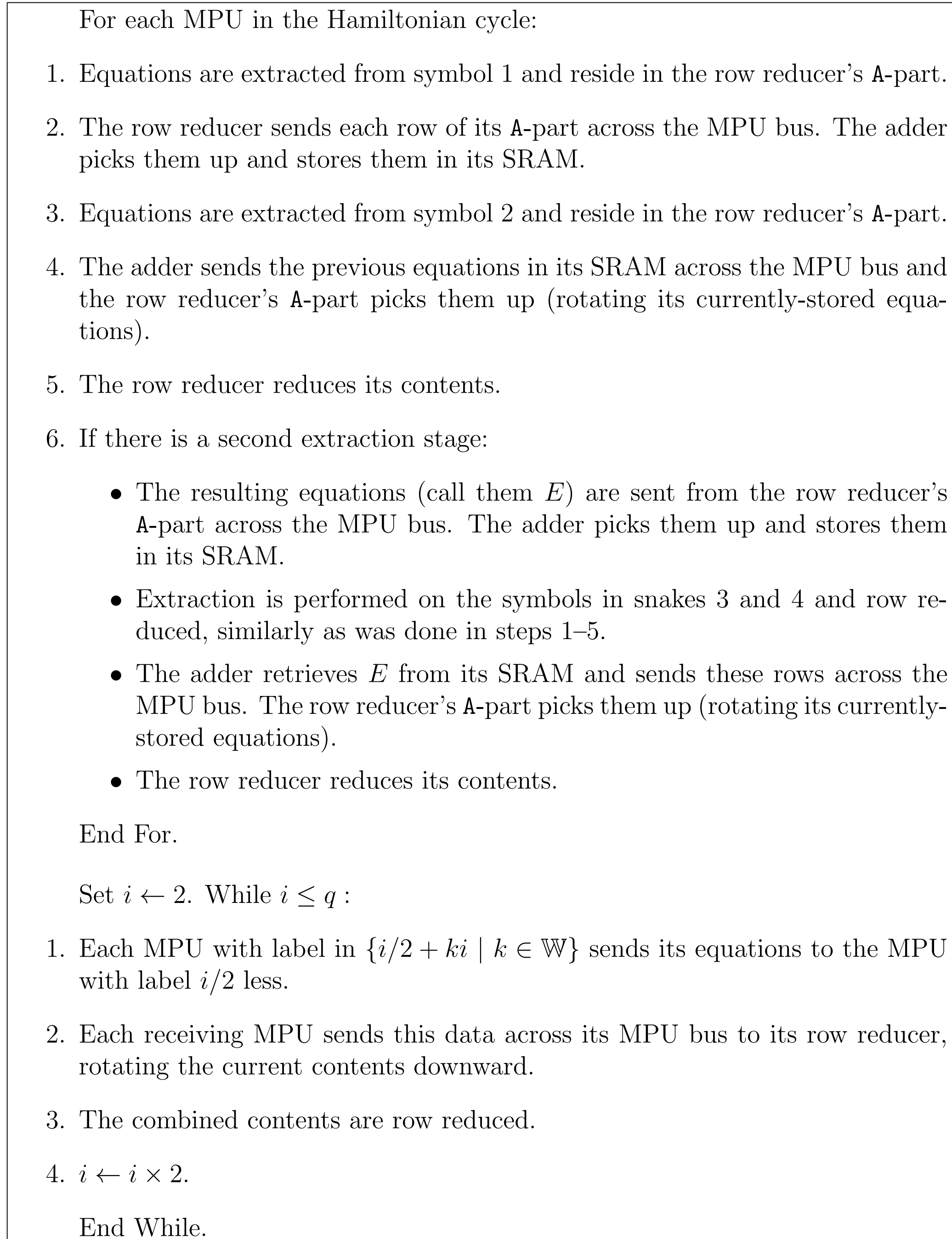
8. For each column d of L_j:

   - d is sent across the MPU bus and the multiplier picks it up.
   - The multiplier sends its s-part s to the adder for adding.
   - If $r \neq 0$, the multiplier sends its r-part to the hash table.
   - If $r \neq 0$, the hash table sends all indices from L_i that match the r-part across the MPU bus to the adder. For each such index i,
     - The s-part at index i is looked up in the adder.
     - The s-part is retrieved, added to s, and sent across the MPU bus.
   - If $r = 0$, the adder runs through all its contents. For each such index i,
     - The s-part at index i is looked up in the adder.
     - The s-part is retrieved, added to s, and sent across the MPU bus.

Figure 4.5: High Level Order of Operations During Gluing

1. The row reducer is reset. (Note that this produces the identity matrix in U.)
2. Starting with the first group of 2^{11} columns of L, for each such group of L:
   - The `A`-part of the row reducer is reset but the `U`-part is preserved.
   - The group of 2^{11} columns of L is sent across the MPU bus to `Us`.
   - For each row of the row reducer's current U:
     - The row reducer sends the left 2^{11} bits of the next row of its current U across the MPU bus and the multiplier's L bus picks it up.
     - The multiplier sends the resulting row across the MPU bus and the row reducer picks it up.
   - The row reducer reduces its contents, modifying the current U.
3. The `A`-part of the row reducer is reset but the `U`-part is preserved.
4. Starting with the first group of 2^{11} columns of L, for each such group of L:
   - The group of 2^{11} columns of L is sent across the MPU bus to `Us`.
   - For each row of the row reducer's U:
     - The row reducer sends the left 2^{11} bits of the next row of U across the MPU bus and the multiplier's L bus picks it up.
     - The multiplier sends the resulting row across the MPU bus and the row reducer's `A`-part picks it up.
   - The row reducer performs zero and one detection on its current `A`-part.
5. The row reducer's `A`-part is reset, preserving its detection flip-flops and `U`.
6. The rows of the symbol's A-part are sent across the MPU bus to `Us`.
7. The multiplier sends the columns of A across the MPU bus to the row reducer.
8. The row reducer sends the columns of A across the MPU bus to `Us`.
9. The `A`-part of the row reducer is reset but the `U`-part is preserved.
10. For each row of the row reducer's U:
    - The row reducer sends the left 2^{11} bits of the next row of U across the MPU bus and the multiplier's L bus picks it up.
    - The multiplier sends the resulting row across the MPU bus and the row reducer's `A`-part picks it up.
11. The row reducer rotates through its `A`-part, setting the 2048th bit of each row according to its detection flip-flops.

Figure 4.6: High Level Order of Operations of Extracting Equations from a Symbol



For each MPU in the Hamiltonian cycle:

1. Equations are extracted from symbol 1 and reside in the row reducer's `A`-part.
2. The row reducer sends each row of its `A`-part across the MPU bus. The adder picks them up and stores them in its SRAM.
3. Equations are extracted from symbol 2 and reside in the row reducer's `A`-part.
4. The adder sends the previous equations in its SRAM across the MPU bus and the row reducer's `A`-part picks them up (rotating its currently-stored equations).
5. The row reducer reduces its contents.
6. If there is a second extraction stage:
   - The resulting equations (call them E) are sent from the row reducer's `A`-part across the MPU bus. The adder picks them up and stores them in its SRAM.
   - Extraction is performed on the symbols in snakes 3 and 4 and row reduced, similarly as was done in steps 1–5.
   - The adder retrieves E from its SRAM and sends these rows across the MPU bus. The row reducer's `A`-part picks them up (rotating its currently-stored equations).
   - The row reducer reduces its contents.

End For.

Set $i \leftarrow 2$. While $i \leq q$:

1. Each MPU with label in $\{i/2 + ki \mid k \in \mathbb{W}\}$ sends its equations to the MPU with label $i/2$ less.
2. Each receiving MPU sends this data across its MPU bus to its row reducer, rotating the current contents downward.
3. The combined contents are row reduced.
4. $i \leftarrow i \times 2$.

End While.

Figure 4.7: High Level Order of Operations During a Mass Row Reduction

4.2.2 Traffic Controller

The traffic controller is a collection of four chips responsible for receiving symbol data from neighbor MPUs, storing it, and pushing it across the MPU bus if need be. After the results of various computations from other functional units are complete, the traffic controller will store or forward to a neighbor MPU those results, depending on what is currently being done. This is the only functional unit that is connected to other MPUs and the MCP, as well as the MPU bus. Details on the architecture of the traffic controller and how it operates are given in Appendix A.

4.2.3 Row Reducer

The row reducer is comprised of a chip named `A/U`, which is connected to the MPU bus. Each part of its name will refer to a separate processing area inside this chip. The row reducer has four responsibilities: compute a row-reduced version of A (i. e., the vertical concatenation of A_i and A_j when they are received), compute the matrix U such that UA yields the row-reduced matrix that will appear in the `A`-part, compute the matrix V such that VL is row reduced, and determine which rows of VA correspond to URHS equations. During a glue, the data stored in the `A`-part will be sent back across the MPU bus. (This corresponds to B in the MRHS gluing algorithm.) During agreeing and gluing, the data stored in the `U`-part will be sent across the MPU bus to the multiplier. During equation extraction, the data stored in both parts will be sent to the multiplier. Details on the architecture of `A` and `U` and how it operates are given in Appendix B. The JONES element used in `A` and `U` builds on ideas from SMITH [5, 6] and may be of independent interest.

4.2.4 Multiplier

The multiplier occupies one part of a chip named `M/HT`. If the MPU is agreeing two symbols, the multiplier receives data from `A/U` and stores it in a processing area called `Ur`. If the MPU is gluing two symbols, the multiplier will also receive additional data from `A/U` and store it in a separate processing area called `Us`. It then receives the L-part of a symbol one column at a time, and multiplies it with the contents in `Ur` and (if gluing) `Us`. Once this multiplication is complete for the received L-column, the multiplier will send the result from `Ur` (called an *r-part*) to the hashtable. If gluing, it will also send the result from `Us` (called an *s-part*) to the adder across the MPU bus. If extracting equations, it will receive data from traffic control or `A/U`, store it in `Us`, receive more data from `A/U`, and send results back to `A/U`. Details on the architecture and working of the multiplier are discussed in Appendix C. Like the row reducer, this architecture might be of independent interest.

4.2.5 Hash Table

The hash table is used in both PET SNAKE's agreeing and PET SNAKE's gluing phase, and it is designed to process one write query per clock cycle—similarly, for look-ups, one look-up query per clock cycle can be coped with. Elements to be stored or looked up in the hash table are r-parts with a (zero padded) size of $r_{max} = 135$ bit, and the hash table is designed to store up to 2^{20} such r-parts. Details on the architecture and the inner working of the hash table are discussed in Appendix D.

Remark. Having no more than 2^{20} columns, identifying each column with a 135 bit hash value seems a safe choice: taking the hash values for being uniformly

distributed, the probability that no collision occurs is $\geq \prod_{i=0}^{2^{20}-1}(1-\frac{i}{2^{135}}) \geq 1-2^{-90}$.

4.2.6 Adder

The adder is comprised of its own chip, which is largely a memory storage device. The adder is only used during gluing and equation extraction. During a glue, while the columns of L_i are being processed, `M/HT` will send out s-parts across the MPU bus. These will be picked up by the adder and stored in a collection of 256 DRAMs. Later, for each column in L_j that is being processed, the adder first acquires an s-part and stores it in a separate row of flip-flops called the *adding register*. Then the hash table will send across the MPU bus either a series of indices in L_i that match to that particular L_j column (i.e., whose Pr_{ij} columns are the same), or a popularity number of the resulting r-part. In the first case, the adder will look up the indices in its DRAM collection. In the second case, it will use the popularity number to find indices in its own table, and look those up in its DRAM collection. The resulting s-parts are then added to the adding register, and the sum is sent back across the MPU bus. During equation extraction, the adder will store groups of equations temporarily to be row reduced later. More details on the architecture and the internal working of the adder are given in Appendix E.

4.3 Performance I: Total Chip Area and Cost

With the area estimates in Appendices A–E, the size of the five functional units per MPU can be summarized as shown in Table 4.1.

Component	Traffic Controller	Row Reducer	Multiplier	Hash Table	Adder
Area in cm^2	4×3.9	3.8	0.43	0.41	1.1

Table 4.1: Size of Individual MPU Components

Thus, the total chip area of the (seven) chips comprising one MPU computes to

$$\underbrace{4 \cdot 3.9}_{4 \text{ chips}} + \underbrace{3.8}_{1 \text{ chip}} + \underbrace{0.43 + 0.41}_{1 \text{ chip}} + \underbrace{1.1}_{1 \text{ chip}} = 21.34 < 22 \text{ cm}^2.$$

For a PET SNAKE architecture with $p^2 = 2^5 \times 2^5$ MPUs, this results into a total chip area of about 2.25 m^2. To enable the necessary wiring, cooling etc. for actually placing the chips (along with the MCP and its agents) some more space will be required. Obviously this is a non-trivial size requirement, but it is important to note that none of the involved chips is larger than 3.9 cm^2, and the resulting device is designed to host a system of symbols as needed to attack a modern block cipher like AES-128.

One MPU uses some 22 cm^2 of silicon. If we assume a 30 cm wafer to cost $5000, the pure silicon for one MPU calculates to about $160. If we apply a factor 4 for the full design, including the board and some safety margin, one MPU is about the price of one PC. Therefore we compare the performance of one MPU with one PC. The next section gives a simplified model to analyze the running time in a software implementation on a PC, and in Section 4.4.5 we present measurements when working with 4 rounds of PRESENT.

4.4 Performance II: PET SNAKE vs. Software

To measure the time cost of an MPU versus software, the MPU's time is measured in clock cycles. For PET SNAKE we assume a 1 GHz clocking rate: with each

component of our architecture having a gate depth of four or less, we believe such a clocking rate not to be implausible. Software's time is given in number of processor *steps*. Factors which relate to the software moving data in and out of memory, cache, and so forth can be captured via a constant α (i. e., each step takes α clocks on average), so a step count serves as a sort of best case scenario for software.

Suppose we are agreeing two symbols S_i and S_j. Let A_i have dimensions $w_i \times y$, A_j have dimensions $w_j \times y$, L_i have dimensions $w_i \times c_i$, and L_j have dimensions $w_j \times c_j$. Note that y then is the number of variables in the cryptosystem. Let β be the number of bits of a value that the processor can perform arithmetic on at once; in modern machines, $\beta \in \{32, 64\}$.

4.4.1 Linear Algebra

Let A be the vertical join of A_i and A_j. Then A has size $(w_i + w_j) \times y$. We suppose that each row will rarely have more than one 1 in A; this is usually true in the middle and later stages of a run. Let γ be the chance a second 1 exists in a column of A provided a 1 exists already in that column. Note that γ will change from symbol to symbol, but $0 \le \gamma \le 1$.

Hardware

JONES has two advantages over software: if a zero column exists, we dispense with it in one step, and if an add is to be performed, this also takes one step. Further, the modifications to U are done in parallel to A.

Let h be the number of columns of A that have more than one 1. Then we have that $\gamma = \frac{h}{w_i + w_j - h}$, and so $h = \frac{\gamma}{1+\gamma}(w_i + w_j)$. Thus, the number of columns of A that have exactly one 1 are $w_i - h + w_j - h$, which yields $\frac{1-\gamma}{1+\gamma}(w_i + w_j)$. Label this value t. Adding h and t gives the total number of populated columns of A. So, if we let z be

the number of columns of A which are all zero, then $y - z = h + t = \frac{1}{1+\gamma}(w_i + w_j)$.

Now, since the matrices A_i and A_j are already row-reduced prior to this process, we have some reasonable expectations on where to find a 1 if it exists in a column at all; that is, if it is not near the main diagonal of A_i, it is near the main diagonal of A_j. It could happen that $h = 0$ and we are extremely unlucky with 1 placement, in which case JONES will take $y + \frac{1}{2}(w_i + w_j)^2$ clocks.

This will almost never happen, however. If there are two ones in the leftmost column of `A`, one of them will be near or at the top. If there is only one 1, it will either be at or near the top, or it will be roughly halfway down. If there are none, we just shiftover without further examining the column. So, for the h columns, we won't have to shift the rows of `A` up, and for about $\frac{1}{2}t$ columns, we still won't. For the other $\frac{1}{2}t$ columns, we can expect to perform shiftups equal to about half of the unlocked rows.

After an add, another locked row is created, so the number of unlocked rows is lessened. Further, we can expect at least two such adds to be performed between times we have to shiftup half of the unlocked rows. Hence, the first time we encounter such a column we shiftup $\frac{1}{2}(w_i + w_j)$ rows, but the next time we encounter such a column we will shiftup $\frac{1}{2}(w_i + w_j - 2) = \frac{1}{2}(w_i + w_j) - 1$ rows. Hence, we have a truncated triangular sum of shiftups to count. Since the number of unlocked rows starts at $w_i + w_j$, we expect a total shiftup count of $\frac{1}{2}(\frac{1}{2}(w_i + w_j))^2 - \frac{1}{2}\left[\frac{1}{2}(w_i + w_j) - \frac{1}{2}t\right]^2$, which yields $\frac{1}{8}\left(1 - \frac{4\gamma^2}{(\gamma+1)^2}\right)(w_i + w_j)$ shiftups. Hence, our total clock count is $y + \frac{1}{8}\left(1 - \frac{4\gamma^2}{(\gamma+1)^2}\right)(w_i + w_j) = y + \frac{1}{8}\frac{(1-\gamma)(1+3\gamma)}{(1+\gamma)^2}(w_i + w_j)$.

Software

Different choices for the algorithm can be made, and here we consider a situation where Gauß elimination is used to perform the row reduction. For the matrix sizes

at hand, this seems a plausible option. Then software must examine w_i+w_j elements in the first column. It first must find a 1, and if successful, it scans the rest of the column looking to add a row. If it finds such a row (i. e., with a 1 in this column), it performs an add of the two rows which takes y/β steps.

It then proceeds to the next column, examining the bottommost $w_i + w_j - 1$ elements, and addition of rows costs $(y-1)/\beta$ steps. And so on. We note that any additions that are performed in A are also performed in the U that is being built, and U has dimensions $(w_i+w_j)\times(w_i+w_j)$, though we do not explicitly count them.

If $y \geq w_i + w_j$, then in total there are $\frac{1}{2}(w_i + w_j)^2$ locations to visit, with a truncated triangular sum of addition steps in A equal to $\frac{\gamma}{\beta}\left[\frac{1}{2}y^2 - \frac{1}{2}(y-(w_i+w_j))^2\right]$ $= \frac{\gamma}{\beta}(w_i + w_j)(y - \frac{1}{2}(w_i + w_j))$. In these cases we expect γ to be closer to 0 than to 1, and so hardware offers at least a factor 4 improvement in clocks over steps.

If $y \leq w_i + w_j$, then we have a truncated triangular sum of locations to visit equal to $\frac{1}{2}(w_i+w_j)^2 - \frac{1}{2}(w_i+w_j-y)^2 = y(w_i+w_j-\frac{1}{2}y)$. The addition steps total $\frac{\gamma}{\beta}\frac{1}{2}y^2$. In these cases we expect γ to be closer to 1 than to 0, and we expect few, if any, zero columns. Hence we use $y = \frac{1}{1+\gamma}(w_i + w_j)$, and putting just the locations expression over the clocks expression, we have a factor improvement equal to

$$\frac{\frac{1}{1+\gamma}(w_i+w_j)\left((w_i+w_j) - \frac{1}{2}\frac{1}{1+\gamma}(w_i+w_j)\right)}{\frac{1}{1+\gamma}(w_i+w_j) + \frac{1}{8}\frac{(1-\gamma)(1+3\gamma)}{(1+\gamma)^2}(w_i+w_j)^2} = \frac{\frac{1+2\gamma}{2+2\gamma}(w_i+w_j)}{\frac{1}{8}\frac{(1-\gamma)(1+3\gamma)}{1+\gamma}(w_i+w_j)+1}$$

As γ increases towards 1, this expression will tend towards a factor $\frac{3}{4}(w_i + w_j)$ improvement (i. e. JONES takes linear time). This does not come as a surprise, for when γ gets closer to 1, there is less and less need to perform shiftups to find 1s.

4.4.2 Matrix Multiplication and Recording Deletions

Hardware

Once `Ur` and `Us` are loaded, their multiplications to L_i occur in parallel; similarly for L_j. Because of the pipeline structure of the multiplier, all the columns of UT_{ij} (similarly, UT_{ji}) are computed at a rate of one clock per column, plus a few clocks of latency in the beginning. The hash table then picks up the resulting r-parts and processes them at a rate of one clock per r-part, and it is also structured in a pipeline fashion.

Hence, processing L_i takes c_i clocks plus a few clocks of latency. Then, processing L_j also takes c_j clocks, plus a few clocks of latency. Since the MPU bus must be used to report a deletion, it will take one clock per deletion, up to a maximum of c_j clocks to report all of L_j's deletions. Finally, L_i is processed again from the hash table's DRAM buffer, and those entries are looked up (for deletions) at the same rate. Since the hash table can report a deletion at the same time as looking up the next value, we count c_i clocks to report any deletions for L_i.

Since the traffic controller can record a deletion in a pipeline fashion and send a column at the same time, no additional overhead is counted for this. Finally, because of the "just in time" nature of symbol transmission, it takes no additional time for a deletion to actually take hold in a symbol.

Thus, two symbols will have their deletions processed in $2c_i + 2c_j$ clocks, plus some small latency. (At the very end of an agreement phase, an additional c_i clocks will also be spent for one symbol. This is a one-time latency cost.)

Software

When constructing Pr_{ij}, using a Method of Four Russians software approach is certainly helpful. The T-storage matrix is set up on each pass. Arranging the data the same way the hardware handles it, this T matrix has 2^k rows of r entries each, where k is the storage constant (typically $k = 8$, but can be increased), and $r = \text{rows}(A) - \text{rank}(A)$. It is built in $2^k \frac{r}{\beta}$ steps. Then, for (the given k bits of) each L_i column, the appropriate entry in the T matrix is read off and stored (taking $\frac{r}{\beta}$ steps), waiting to be added later. This continues for the entire pass. Hence, a pass takes $2^k \frac{r}{\beta} + c_i \frac{r}{\beta}$ steps. Afterwards, a new T matrix will need to be built. Since there are $\frac{w_i}{k}$ passes, all passes total comprise $\frac{w_i}{k}(2^k + c_i)\frac{r}{\beta}$ steps.

After all passes are complete, the subresults are added together to produce the final result of the multiplication. We can use $\log \frac{w_i}{k}$ additions of matrices, each addition taking $c_i \frac{r}{\beta}$ steps. This gives a total step count of

$$\frac{w_i}{k}(2^k + c_i)\frac{r}{\beta} + c_i \frac{r}{\beta} \log \frac{w_i}{k} = \frac{r}{\beta}\left(\frac{w_i}{k}2^k + c_i(\frac{w_i}{k} + \log \frac{w_i}{k})\right)$$

to construct Pr_{ij}. A similar expression will result when constructing Pr_{ji}.

One could try to optimize by increasing k to 16 or so, but $k = 32$ is troublesome as the 2^k term starts to dominate.

The situation gets worse for software; it still has to search through the data to find matching r-parts. Sorting Pr_{ij} will take at least $c_i \log c_i$ steps and as many as $\frac{r}{\beta} c_i \log c_i$, should many r-parts become popular. Similar expressions result when sorting Pr_{ji}. Finally, a bilinear search taking $\frac{r}{\beta}(c_i + c_j)$ more steps must be performed to find matching r-parts. Once the mismatches are found, columns have to be deleted from L_i and L_j; this takes $\frac{r}{\beta}(c_i + c_j)$ steps. Hence, total sorting and searching for both matrices takes $\frac{r}{\beta}\left(c_i(2 + \log c_i) + c_j(2 + \log c_j)\right)$ steps.

In total, we have

$$\frac{r}{\beta}\left(\frac{w_i + w_j}{k}2^k + c_i(\frac{w_i}{k} + 2 + \log c_i \frac{w_i}{k}) + c_j(\frac{w_j}{k} + 2 + \log c_j \frac{w_j}{k})\right)$$

steps to agree the symbols S_i and S_j.

The MPU has a very clear and obvious advantage. Aside from the additional terms the software induces in its step count, it is important to stress that the hardware does not rely on the values of r, w_i, or w_j at all. Hence, large r (whose maximum value is 2^{11}) will dramatically slow down the software, but the hardware will be unaffected. Since r will steadily increase over the entire run, hardware's advantage will grow over time.

4.4.3 Gluing

Both hardware and software must pay the linear algebra times and the multiplication times as described earlier. From there the situation changes slightly. At this point we know that we may only construct a symbol whose L-part has no more than 2^{20} columns, so we label the number of such columns d.

Hardware

During the matrix multiplication of L_i, r-parts are being stored in the hash table at the same time, so we do not count this cost again. However, s-parts are being sent to the adder at the same time, so the adder's DRAM collection is filled for free.

Afterwards, the hash table will go through a preprocessing of its c_i entries. It may happen that these values hit the SRAM of the adder entirely too quickly, at which point we must pay upwards of an 8-clock penalty per such index. In the worst case this takes $8c_i$ clocks in total, but is expected to average to more like $2c_i$ over

the course of an entire run.

Then, L_j is processed. We get an s-part in one clock (after some latency), and at the same time, its r-part is examined for matches in the hash table. If the hash table has the matching indices, it simply sends them, one per clock. If the adder has them, the adder uses its SRAM to produce them to the s-lookup chain. Since the SRAM produces values 128 bits at a time (that is, 6 indices per 8 clocks), the penalty of multiple fast read requests is mitigated.

Hence, we have worst case behavior of $8c_i + \frac{8}{6}d$ and best case behavior of $c_i + d$ clocks to finish all additions.

Software

It is plain that the software will suffer tremendously if it has to re-match r-parts to find corresponding s-parts to add, so we give it a fighting chance by allowing it to store the matching indices during agreement. (This gets expensive in memory with a state of several hundred symbols, but can nonetheless be theorized.)

Then it merely performs lookups of its storage data. Since there are d pairs of s-parts to be added, software takes $\frac{r}{\beta}d$ steps to finish all additions. Again, as r steadily increases over a run, software becomes vastly inferior to hardware, which does not rely on the value of r.

4.4.4 Equation Extraction

We begin by analyzing the time taken by extracting equations from a particular symbol with A of dimensions $w \times y$ and L of dimensions $w \times c$. We suppose A has the same bias of data as described in Section 4.4.1, but L is not guaranteed to have any bias of data. We calculate supposing that L's 0s and 1s are uniformly distributed.

Hardware

We follow Figure 4.6. In step 1, the row reducer is reset, taking 4096 clocks to bring `U` back to the identity matrix. Then we have $\lceil c/2^{11} \rceil$ groups of columns of L to process to find U such that UL is row reduced. For each of these groups, we first send the 2^{11} columns to `Us`, taking 2^{11} clocks, followed by sending the top 2^{11} rows of the row reducer's `U`-part, each producing a row that the `A`-part must store. Each row takes two clocks (one to read, one to write, as data must go back and forth across the MPU bus). So, to get a temporary result of a multiplication in `A`, we require 2^{12} clocks. To rotate `U` back into position, we require another 2^{11} clocks.

Then `A` gets row reduced, modifying the current `U`. Because L's bits are uniformly distributed, UL's bits will be also, and JONES will behave at least as well as SMITH under these conditions. Since it has been reported that SMITH will take $2k$ time for such a $k \times k$ matrix [6], JONES will take at most 2^{13} cycles to row reduce `A`. In total, step 2 takes $\lceil c/2^{11} \rceil (2^{11}+2^{12}+2^{11}+2^{13})$ clocks, which is at most $c/2^{11} \times 8(2^{11}) = 8c$ clocks.

Step 3 takes at most 2^{12} clocks, since we just need to reset `A`. In step 4, we again have $\lceil c/2^{11} \rceil$ groups of columns of L to process. For each group, we first send it to `Us` taking 2^{11} clocks. Then the multiplication happens once more, taking 2^{12} clocks, with the temporary result in `A`. Then zero and one detection commence, requiring `A` to cyclically shift upwards completely, taking 2^{12} clocks. In the first 2^{11} of these, the ZD column is populated, and the OD row gets set to the sum of all rows in `A`. Then in the second 2^{11} clocks, the OD row cyclically shifts left, setting the OD flag. Hence, step 4 takes $\lceil c/2^{11} \rceil (2^{11} + 2^{12} + 2^{12})$, which is at most $5c$ clocks.

Step 5 is similar to step 3, taking 2^{12} clocks. Step 6 takes w clocks to populate `Us`. Step 7 takes at most 2^{12} clocks (one to multiply, one to send) to send the columns

of A back to `A`. Step 8 takes 2^{11} clocks to repopulate `Us`. Step 9 is similar to step 5, taking 2^{12} clocks. Step 10 will require 2^{12} clocks (one to send, one to receive the multiplication, for each row in `U`). Step 11 will require 2^{12} clocks to set the 2048th element according to its detection flip-flops, followed by another 2^{12} clocks to put the (potentially) nonhomogeneous equation at the top.

Hence, to extract the equations from a symbol, PET SNAKE uses at most $2^{12}+2^{11}+8c+2^{12}+5c+2^{12}+w+2^{12}+2^{11}+2^{12}+2^{12}+2^{12}+2^{12} = 18(2^{11})+13c+w \leq 13,670,400$ clocks.

Software

We once again consider Gauß elimination for the row reduction. In almost all cases $w \ll c$, and since each entry of L is equally likely to have a 0 or a 1, we note it will take one or two steps to find a pivot row for row i. However, once a pivot row is found, it will have to be added to about half the remaining rows, and each such addition will take $\frac{c-i}{\beta}$ steps. Hence, the step count is

$$\sum_{i=1}^{w} \frac{w-i}{2}\frac{c-i}{\beta} = \frac{w}{4\beta}\left[cw - \frac{1}{3}w^2 - c + \frac{1}{3}\right]$$

which is easily dominated by the $\frac{cw^2}{4\beta}$ term. As the run continues, w approaches y, and c almost always remains at 2^{20}. Taking an average value of w to be 2^{10} and $\beta = 32$, this term becomes 2^{33}.

Once L is row reduced, we must take the corresponding U (of size $w \times w$) and multiply it to A. The cost for this is negligible, though, using the Method of Four Russians again. Each T matrix costs $2^k\frac{y}{\beta}$ to set up, reading off the correct row costs $\frac{y}{\beta}$ steps, so each pass takes $\frac{y}{\beta}(2^k + w)$ steps. There are $\frac{w}{k}$ passes, giving a step count of $\frac{wy}{k\beta}(2^k + w)$ to construct all $\frac{w}{k}$ matrices to be added. We can structure things to

take $\log \frac{w}{k}$ additions, each addition costing $\frac{wy}{\beta}$ steps, for a total of

$$\frac{wy}{\beta}\left(\frac{1}{k}(2^k + w) + \log \frac{w}{k}\right)$$

steps for the entire multiplication. However, using the same values as above (with $y = 2^{11}$ and $k = 8$), this reduces to approximately 2^{23} steps.

We see that the cost in software is about a factor of 1000 in steps over clocks for the equation extraction in the common case.

Assigning the final 0/1 column to construct the equations is trivial in both settings. Software provides no benefit over hardware when bringing all the equations together to be row reduced, so we do not perform an analysis of this. Finally, reducing with the current equation set to determine consistency is also trivial in both settings.

4.4.5 Software Measurement

It should be noted that, in the above derivations, the linear algebra is almost always dominated by matrix multiplication and recording deletions, both in hardware and in software.

In order to get a handle on performance metrics, four rounds of PRESENT were cryptanalyzed in software ($k = 8$, $y = 308$) using MRHS with the above options, and this entire session's timing values were recorded. The platform was an Intel E2180 processor, $\beta = 32$, on a single core of 2 GHz, with 2 GByte of RAM. Out of the nearly 10,000 agreements that took place, the vast majority took less than two seconds. We removed these from consideration since fractions of seconds were not measured. Many calculations were made on the remaining 350 or so agreements using the above step count formulas, some results of which are

illustrated in Table 4.2. We see no problem using just these ∼350 values since in a full cryptosystem operated on by PET SNAKE, there will commonly be high w_i, w_j, r, c_i, and c_j values, and these data points are more reflective of this scenario. It should be noted that we calculated steps using $\gamma = 0.5$; varying γ in either direction does not adversely affect our overall results.

An average of the ∼350 time improvement factors gives an average improvement of 2,281 for four rounds of PRESENT. As noted above, as r gets larger, we suspect PET SNAKE will only improve from there.

To get a better feeling for just how much more favorable PET SNAKE will be, we see that an average of the ∼350 α data points gives $\alpha = 66.068$, where α is the metric of steps per processor clock. Some things are not included in the step count, such as loop counter variables incrementing, allocation space instructions, and low-level memory management.

Once we have a good handle on the α that a given processor exhibits, we can predict software behavior for larger systems. For example, if PET SNAKE runs an MRHS attack on AES-128 or more rounds of PRESENT, it won't be uncommon for $y > 1500$, $w_i > 1024$, and $r > 1024$. Modeling such systems in software directly is problematic owing to the lack of sufficient on-board memory at the time of this writing, but we can predict step counts for software under these conditions. Table 4.3 gives the relevant predictions fixing $\alpha = 66.068$. In the later stages of a given attack of a full cipher of something like AES-128,we'll see symbol sizes listed in this table. The relative improvement of PET SNAKE is now even clearer, touching a six-digit improvement.

Finally, it is worth noting that other software methods may be used to multiply large matrices; it is certainly possible that some of them may be more efficient than the Method of Four Russians, and so the improvement factor may be reduced.

w_1	c_1	w_2	c_2	r	time (s)	total steps	Pent clks	α	PS clocks	PS time (s)	improvement
208	32768	236	524144	192	4	185372686.4	$8 \cdot 10^9$	43.1563	1127822	0.001127822	3546.658959
211	98304	236	196796	192	2	94214489.28	$4 \cdot 10^9$	42.4563	604383.625	0.000604384	3309.156498
211	98304	236	524144	192	4	205688510.1	$8 \cdot 10^9$	38.8937	1259079.625	0.00125908	3176.923779
229	121856	236	196796	192	2	103100923.9	$4 \cdot 10^9$	38.7969	652627.625	0.000652628	3064.534695
213	14336	237	196796	190	2	68614875.44	$4 \cdot 10^9$	58.2963	436634.5	0.000436635	4580.49009
207	32768	229	248832	190	3	89847645.61	$6 \cdot 10^9$	66.7797	576709.1111	0.000576709	5201.929261
207	32768	229	248832	190	2	89847645.61	$4 \cdot 10^9$	44.5198	576709.1111	0.000576709	3467.95284
217	16384	229	248832	189	2	85273467.77	$4 \cdot 10^9$	46.9079	544553.6111	0.000544554	3672.732967
212	16384	229	497664	189	3	168450194.4	$6 \cdot 10^9$	35.6188	1041909.625	0.00104191	2879.328425
212	16384	229	786432	189	5	266482279.2	$10 \cdot 10^9$	37.5259	1619445.625	0.001619446	3087.476308
213	28672	236	524144	189	3	184351734.1	$6 \cdot 10^9$	32.5464	1119940.069	0.00111994	2678.714765
213	57344	229	497664	189	3	180808589.4	$6 \cdot 10^9$	33.1842	1123890.944	0.001123891	2669.298133
210	98304	229	248832	189	2	110163469.2	$4 \cdot 10^9$	36.3096	707963.4028	0.000707963	2825.004784
210	98304	229	497664	189	3	193446733.7	$6 \cdot 10^9$	31.0162	1205627.403	0.001205627	2488.330966
212	12288	229	248832	187	2	83962706.02	$4 \cdot 10^9$	47.6401	536053.625	0.000536054	3730.970013
212	12288	229	786432	187	5	265278055.2	$10 \cdot 10^9$	37.6962	1611253.625	0.001611254	3103.173779
213	57344	229	497664	185	3	180808589.4	$6 \cdot 10^9$	33.1842	1123890.944	0.001123891	2669.298133
213	57344	229	786432	185	5	278840674.1	$10 \cdot 10^9$	35.8627	1701426.944	0.001701427	2938.709779
213	16384	236	524144	184	3	180691970.9	$6 \cdot 10^9$	33.2056	1095364.069	0.001095364	2738.815416
134	1048576	147	131072	102	5	202949079.1	$10 \cdot 10^9$	49.2734	2365087.403	0.002365087	2114.086775
125	4096	141	450816	101	2	78081960.56	$4 \cdot 10^9$	51.2282	915045.6111	0.000915046	2185.683397
125	4096	137	1048576	101	4	185889011.7	$8 \cdot 10^9$	43.0364	2110418.944	0.002110419	1895.35827
122	8192	137	1048576	101	4	186478052	$8 \cdot 10^9$	42.9004	2118502.403	0.002118502	1888.126251
129	16384	137	524288	101	2	93036403.87	$4 \cdot 10^9$	42.9939	1086565.611	0.001086566	1840.661972
129	65536	140	450816	101	3	87773276.82	$6 \cdot 10^9$	68.3579	1038037.069	0.001038037	2890.070199
129	65536	137	1048576	101	6	195580582.3	$12 \cdot 10^9$	61.3557	2233445.611	0.002233446	2686.432107
135	1048576	152	131072	101	5	202950785.3	$10 \cdot 10^9$	49.2730	2365324.069	0.002365324	2113.875246
134	1048576	139	1048576	101	7	366057846.9	$14 \cdot 10^9$	38.2453	4199787.625	0.004199788	1666.750947

Table 4.2: Some Measured Values of Software Performance ($k = 8$, $\beta = 32$, $\gamma = 0.5$, $y = 308$)

However, PET SNAKE's time is still unaffected by these large symbols, processing each pair in less than half of a hundredth of a second. We feel that such absolute speed is too compelling to be dismissed.

4.5 Performance III: Parallelization

Guessing Variables

PET SNAKE will, in its depth-first search of keys, eventually guess enough keys so that either the system is found to be inconsistent or the key is correct. This number of keys we refer to as δ. So that it may make appropriate use of parallelism, PET SNAKE will eventually guess enough keys discovering δ, and then make note of its available storage. Then the MCP will be able to determine how high in the guess tree it can fork a new guess into another area of the board, while having the ability to store the states required for a sub-branch of this new guess as well as for the original branch. The idea here is that PET SNAKE will use all of its MPUs to finish off a branch of a guess tree as quickly as possible. If more MPUs become available, more guesses can potentially be forked.

Should the MCP determine that storage will run out, it will delete some states higher in the guess tree. Any such state which needs to be recovered later can always be recalculated based on the next-highest state in the guess tree, and the remaining key guess symbols to affect the deleted state's guess. It is true that these (possibly several) guesses will need to be re-performed in one series of agrees and glues, increasing the overall running time, but PET SNAKE at least has recovery options should storage requirements vary wildly across parallel branches of the guess tree. For this reason, PET SNAKE will never delete the state which was arrived at before any guesses were committed (i.e., the base state).

w_1	c_1	w_2	c_2	r	time (s)	total steps	pentium clks	PS clocks	PS time (s)	improvement
1000	1048576	1000	1048576	500	170.7644319	5169289689	$3.41529 \cdot 10^{11}$	4474082	0.004474082	38167.48115
750	1048576	750	1048576	300	84.95722911	2571779869	$1.69914 \cdot 10^{11}$	4352054	0.004352054	19521.17991
500	1048576	1000	1048576	200	59.44129911	1799375263	$1.18883 \cdot 10^{11}$	4352304	0.004194304	14171.91007
500	1048576	1000	1048576	400	110.3584658	3340712542	$2.20717 \cdot 10^{11}$	4352304	0.004194304	26311.50861
500	1048576	1000	1048576	600	161.2756325	4882049821	$3.22551 \cdot 10^{11}$	4352304	0.004194304	38451.10715
500	1048576	1000	1048576	800	212.1927992	6423387100	$4.24386 \cdot 10^{11}$	4352304	0.004194304	50590.70569
1500	1048576	1500	1048576	1000	482.5320576	14606952758	$9.65064 \cdot 10^{11}$	4821304	0.004194304	115044.6075

Table 4.3: Some Projected Values of Software Performance ($k = 8$, $\beta = 32$, $\gamma = 0.5$, $y = 308$, $\alpha = 66.068$)

Using Multiple PCs

To cope with a cipher like AES-128, the only plausible option seems to use a cluster of PCs, but here the communication cost between these PCs will add another significant factor to the overall running time of the algorithm. Connecting networked PCs in the same way as PET SNAKE connects its MPUs will introduce additional time spent: suppose that a grid of PCs is connected so each can talk to its neighbor in each cardinal direction using gigabit Ethernet, and suppose that this network actually communicates perfectly (i. e., 1 gigabit/sec). PET SNAKE's connections are 1024 wires clocked at 1 GHz, so it can transmit 1000 gigabit/sec between MPUs. This makes the PC network 1000 times as slow. With the observation that a PC agrees $\gg$1000 times slower than an MPU, the PCs could also implement a 'just in time' delivery method to reduce agreement communication times. However, when symbols need to be moved between agreement stages or to prepare for a glue, we see that the movement time for a PC is a little over 2.15 sec per symbol per hop (over 4.5 minutes per agreement phase, assuming no deletions), whereas for PET SNAKE it is 0.00215 sec per symbol per hop. Hence, a faster network between PCs will need to be established, which in turn adds to the cost of such a solution.

Finally, for multiple PCs to provide the same storage as PET SNAKE, a single PC has to store 4680 MB, not including active memory of at least 325 MB. This is slightly larger than 4 GB per PC, and so more expensive motherboards that can provide larger memory will need to be acquired. (Slower storage solutions like hard drives can be used instead, but given their notorious relative slowness, the times for loading and storing would start to dominate an overall time estimate, and this would make finding a key infeasible.)

Chapter 5

The Transfer Formula

In this chapter, we lay out a process to predict the number of clocks PET SNAKE will take in attacking a block cipher. At the time of this writing, the necessary theoretical tools are not available to predict MRHS's running time over arbitrary cryptosystems. However, we develop a process which works with data generated from a Clump Search software run of MRHS to produce expected running times for PET SNAKE. As theoretical tools become available, parts of this process can be replaced with their improved versions until we do not have to rely on data generated from a software run anymore. Underpinning this effort are two simplifying assumptions, which we discuss shortly.

5.1 PET SNAKE's Guess Tree

We observe that PET SNAKE will have no *a priori* knowledge of the key values, so it will perform a depth-first search, and the software values in Section 3.2.5 suggest that guessing linearly will be most advantageous for threshold 20. Hence we can imagine its guess tree, whose nodes symbolize points where the state was saved, and whose branches are the turns where a guess was committed much like in Section

3.2.3. However, there are some differences between PET SNAKE's guess tree and a software guess tree.

After the initial turn, the number of symbols in any MRHS attack can drop dramatically. Since PET SNAKE seeks to split the MPUs in such a way that many guesses (of the same variables) are performed simultaneously, we must take this into account when calculating clock cycles. Table 5.1 lists the number of key variables that PET SNAKE will guess simultaneously and the number of active areas (one for each guess) that the platform will split into after the base state is reached.

symbols	active areas	MPUs in each active area	bits to guess
1025-4096	1	1024	1
513-1024	2	512	1
257-512	4	256	2
129-256	8	128	3
65-128	16	64	4
33-64	32	32	5
17-32	64	16	6
9-16	128	8	7
5-8	256	4	8
1-4	512	2	9

Table 5.1: Bits to Guess After Base State is Reached

For example, if the base state has 140 symbols, then PET SNAKE will guess three key variables simultaneously and split itself into 8 active areas: one for the guess 000 of these three variables, one for the guess 001, one for the guess 010, and so on. Note that if the symbol count is higher than 1024, then PET SNAKE will still guess one bit, but no splitting will occur. Also note that the smallest active area is two MPUs, owing to the fact that the symbols have to move to another MPU to commit deletions in an agreement. After subsequent states are reached, it is extremely unlikely that the symbol count will drop dramatically (through two powers of 2), so PET SNAKE's guess tree will look a lot like a software run's guess

tree for these states. To ease discussion, we define some basic terminology below.

Definition 5.1. *The* guess depth *of a turn is the number of guesses that have been committed prior to that turn being executed.*

Hence, the guess depth is just the depth of where we are in the guess tree. The initial turn is said to have a guess depth of zero. Note that for software using Linear Choice, the guess depth of a turn is equal to the number of variables that have been guessed. However, for PET SNAKE, the situation may be different since several variables might be guessed at once after the base state is reached; hence, a turn with guess depth 2 might have nine variables guessed already.

Finally, we are in position to describe the first of our simplifying assumptions.

Assumption 5.1. *The number of deletions recorded during all agreement phases during one turn constitutes an average of the number of deletions during all of the agreement phases for any turn of that guess depth.*

It may happen that turns of different guess depths (that is, turns where different numbers of variables have already been guessed) will have a different number of deletions recorded during their agreement phases, but we suppose that all turns with the same depth will have an average number of deletions very close to what we recorded.

Without this assumption, software runs would have to produce deletion counts over each turn in the entire guess tree, but as we've seen, such software runs would be timewise prohibitive for the ciphers we are interested in. (Indeed, if we could have feasibly performed a depth-first search in software, we wouldn't have designed PET SNAKE in the first place.) We shall return to this when we discuss multiplier effects in Section 5.3.1.

5.2 The Deletion Model

The most difficult part of predicting runtimes is taking into account which symbols, when agreed, induce deletions of columns in L-parts. Even if we tracked through a software run which symbols had deletions at which times, trying to correlate this to PET SNAKE's process is rather difficult: movements of these symbols through MPUs is hard to track, and renumbering effects will place symbols in different snakes. Further, even if we tracked how precisely the symbols moved and behaved for a particular PET SNAKE run, there's no guarantee that they would have the exact same behavior across different turns of the same guess depth. So, we turn to a probabilistic analysis to get a handle on how a given number of deletions in a given number of active-area MPUs get processed.

5.2.1 Rooted Trees

Since some deletions cannot occur until other deletions occur first, we choose to model the deletions as a collection of *rooted trees*, which are not to be confused with the guess trees described earlier. In each tree, each node symbolizes a deletion after two symbols are agreed, and each child of a node symbolizes deletions that can now occur as a result of the parent node's deletions taking place. In the beginning of an agreement phase, it is certainly possible that many deletions do not depend on each other, so these deletions are the roots of the trees in this collection. Such a collection of trees is sometimes referred to as a *rooted forest*.

Much information about rooted trees is available in [40, sequence A000081], and some of those facts will be used shortly. We observe that the order of subtrees of a given node is irrelevant; it does not matter which subtree is the first subtree, which is the second, and so on, since we use the tree to simply model which deletions

depend on which. Hence the choice of a rooted tree is appropriate; we view a rooted tree as an equivalence class of trees, where two trees are equivalent if one can be transformed into the other by re-ordering subtrees [33]. Similarly, a rooted forest is an equivalence class of forests, where two forests are equivalent if one can be transformed into the other by re-ordering the rooted trees. (Alternately, we may consider a rooted forest as nothing more than the subtrees of a rooted tree whose root is hidden.)

As software performs an MRHS attack, it is not easy to determine which deletions were dependent on previous deletions and which were independent. Instead, we record the number of deletions committed in each agreement phase, along with the sizes of the symbols in the state before each segment. To determine the structure of the rooted forest corresponding to the deletion dependencies, we make our second simplifying assumption that it is similar to a rooted forest taken uniformly at random.

Assumption 5.2. *The distribution of rooted forests corresponding to m deletions in all MRHS runs is similar to a uniform random distribution of rooted forests of m nodes.*

What was not previously known was how, for any $m \in \mathbb{N}$, to generate a rooted forest of m nodes uniformly at random in a reasonable way. If one relaxes the equivalence condition and merely examines forests, then a uniform random generation algorithm is obtained [7] by modeling them using a context-free grammar for use in a genetic algorithm. However, it is not clear that it is possible to create *rooted* trees and forests using a context-free grammar, so we do not use this algorithm. We instead develop a different algorithm which, as it happens, shares a lot of features with the one in [7]. Later we shall execute this algorithm for $m \leq 1000$.

5.2.2 Generating Rooted Forests Uniformly at Random

Suppose we wish to uniformly at random generate a rooted forest of m nodes. We first construct some data tables dynamically (so that no unneeded space is allocated), and then we perform many lookups on those tables. We begin with some notation.

Definition 5.2.

i) T_m *is the set of rooted trees of* m *nodes.*

ii) F_m *is the set of rooted forests of* m *nodes.*

This process has a side-effect of implicitly constructing two families of bijections t and f where $t_m : \mathbb{Z}_{\#T_m} \xrightarrow[\text{onto}]{1-1} T_m$ and $f_m : \mathbb{Z}_{\#F_m} \xrightarrow[\text{onto}]{1-1} F_m$. As such, we define an ordering on trees as follows: for any two trees $u, v \in T_m$, $u < v$ if and only if ${t_m}^{-1}(u) < {t_m}^{-1}(v)$. In light of the above observation about m-node rooted forests being $(m+1)$-node rooted trees, once one family of bijections is constructed, so is the other. We note immediately that this ordering on rooted trees will not be the same as that in [33], since we will sort subtrees first by node count as described below.

We look at a rooted forest of m nodes as being constructed by a collection of r rooted trees a_1, a_2, ..., a_r, for some $r \in seg_m$, with respective nonincreasing node counts c_1, c_2, ..., c_r such that $\sum c_i = m$.

We then construct sequences of counts b_i such that $b_{11}, b_{12}, \ldots, b_{1s_1}$ are the s_1 counts starting with c_1 that are equal to c_1, and $b_{21}, b_{22}, \ldots, b_{2s_2}$ are the s_2 counts starting with c_{1+s_1} that are equal to c_{1+s_1}, and so on, for d sequences. This breaks up the counts into subsequences of equal-valued terms. For example, if the counts c were 9, 8, 8, 8, 7, 7, 6, 4, 3, 3, 3, 3, 2, 1, 1, 1, then b_1 has one term (namely 9), b_2

has three terms (all of which are 8), b_3 has two terms (both of which are 7), b_4 has one term (namely 6), and so on, ending with b_8 having three terms (all of which are 1) and $d = 8$.

Since we envision the trees T_k (for any $k \in \mathbb{N}$) as being ordered, for each $i \in seg_d$ we must count the number of ordered arrangements of s_i trees in $T_{b_{i1}}$. Call this number B_i. We then calculate the number of rooted forests with this count sequence as $\prod B_i$. In order to correlate a number in $\mathbb{Z}_{\#F_m}$ to a forest in F_m, we must have a way to obtain the number of forests of subtrees with any nonincreasing count sequence. As one might imagine, this is done recursively using the building blocks described below.

Setup

The setup phase of the algorithm consists of building three tables.

First, for each $i \in seg_m$, $\#T_i$ is calculated using the recurrence formula

$$\#T_i = \frac{1}{i-1} \sum_{k=1}^{i-1} \left[\left(\sum_{d \mid k} d \cdot \#T_d \right) \#T_{i-k} \right]$$

with $\#T_1 = 1$ [40]. This takes $O(i^2)$ time for each i, totalling a time of $O(m^3)$.

Then an $m \times m$ table R called the *runtable* is created. Its purpose is to store forest counts in the following way: for any two $i, j \in seg_m$, R_{ij} is the number of sequences $u : seg_j \to T_i$ such that $u_1 \leq u_2 \leq \cdots \leq u_j$; in other words, it is the number of nondecreasing j-length sequences of i-node rooted trees.

To calculate these values, we take advantage of the following theorem:

Theorem 5.1.

$$(\forall n \in \mathbb{N})(\forall k \in \mathbb{W}) \left[\sum_{i=1}^{n} \binom{i+k}{k+1} = \binom{n+k+1}{k+2} \right]$$

Proof

This is a reformulation of the parallel summation formula in [18, p. 174]. ■

We now make an observation about sequences of finite length.

Theorem 5.2. *For each $i, j \in \mathbb{N}$, let S_{ij} be the number of nondecreasing sequences $u : seg_j \to seg_i$. Then*

$$(\forall j \in \mathbb{N})(\forall i \in \mathbb{N}) \left[S_{ij} = \binom{i+j-1}{j} \right].$$

Proof

We proceed by induction on "j". Let $\varphi(j)$ be the formula $(\forall i \in \mathbb{N})[S_{ij} = \binom{i+j-1}{j}]$ for each $j \in \mathbb{N}$.

$\varphi(1)$: Let $i \in \mathbb{N}$. Then S_{i1} is the number of nondecreasing sequences $u : seg_1 \to seg_i$. But there are only i such things, one for each choice of u_1. Hence, $S_{i1} = i = \binom{i}{1} = \binom{i+1-1}{1}$. Thus, $\varphi(1)$.

Let $k \in \mathbb{N}$ and assume $\varphi(k)$: $(\forall i \in \mathbb{N}) \left[S_{ik} = \binom{i+k-1}{k} \right]$.

$\varphi(k+1)$: Let $i \in \mathbb{N}$. Consider $S_{i(k+1)}$. These are all the nondecreasing sequences $u : seg_{k+1} \to seg_i$. Now let us consider the possibilities for u_1. If $u_1 = 1$, then the remaining terms comprise a k-length nondecreasing sequence to seg_i. But the number of such sequences is just S_{ik}. Now, if $u_1 = 2$, the remaining terms comprise

a k-length sequence to $seg_i - 1$. But the number of such sequences is the same as the number of k-length sequences to seg_{i-1} (just subtract 1 from each term), which is $S_{(i-1)k}$. Similarly, for each $v \in seg_i$, if $u_1 = v$, then the remaining terms comprise a k-length sequence to $seg_i - seg_{v-1}$, whose count is the same as the count of k-length sequences to $seg_{i-(v-1)}$ (by subtracting $v - 1$ from each term), which is $S_{(i-v+1)k}$. Hence,

$$
\begin{aligned}
S_{i(k+1)} &= S_{ik} + S_{(i-1)k} + \cdots + S_{1k} \\
&= \sum_{v=1}^{i} S_{vk} \\
&= \sum_{v=1}^{i} \binom{v+k-1}{k} && \text{by Ind. Hyp.} \\
&= \binom{i+k}{k+1} && \text{by Theorem 5.1} \\
&= \binom{i+(k+1)-1}{k+1}.
\end{aligned}
$$

Thus, $\varphi(k+1)$. The rest follows by the Principle of Mathematical Induction and routine steps. ■

Since, for each $j \in seg_m$, R_{ij} is the number of nondecreasing sequences from seg_j to T_i, this is the same as the number of nondecreasing sequences from seg_j to $seg_{\#T_i}$, which is $\binom{\#T_i+j-1}{j}$ by Theorem 5.2. Hence, we build the R table by populating it with this binomial coefficient for each $i \in seg_m$ and $j \in seg_m$ such that $j \leq \lfloor \frac{m}{i} \rfloor$; j is restricted in this way since, for any choice of tree size i in an m-node forest, you can only have at most $\lfloor \frac{m}{i} \rfloor$ such trees. As a side effect, we see that $\#T_i$ is stored in R_{i1} by this process.

As a point of interest, note that we had to use this binomial simplification when populating the R table. Otherwise, since $\#T_i$ is asymptotically $0.4399 \cdot 2.9558^i \cdot i^{-3/2}$ [40, sequence A000081], asking the computer to perform the sum listed in the proof of Theorem 5.2 would become infeasible very quickly.

We construct two more tables, the two-dimensional *partable* denoted P, and the three-dimensional table *lentable* denoted L. For each $i, j \in seg_m$, P_{ij} is the number of rooted forests of i nodes whose first tree has j nodes. It could be that the first few trees have j nodes, so we keep track of this using the lentable: L_{ijk} is the number of rooted forests of i nodes whose first k trees have j nodes. To calculate L_{ijk}, we must first add up $P_{(i-jk)q}$ for each $q \in seg_{j-1}$, and multiply this to R_{jk}; if $i - jk = 0$, we just use R_{jk}. Then,

$$P_{ij} = \sum_{k=1}^{\lfloor \frac{i}{j} \rfloor} L_{ijk}.$$

Finally, we recognize that $\#F_m = \#T_{m+1}$ gives us no intuitive breakdown of all the counts, but

$$\#F_m = \sum_{j=1}^{m} P_{mj}$$

does. Note that, though we concern ourselves with how a given number of nodes m breaks down into each partition of m, this setup prevents us from having to loop through each partition of m, which also would be infeasible very quickly.

We remark that the storage for R is $O(m^2)$ but is significantly less than m^2 since, for each $i \in seg_m$, we only populate R_{ij} when $j \leq \lfloor \frac{m}{i} \rfloor$. Further, for similar reasons the storage for L is $O(m^3)$, but is significantly less than m^3.

Teardown

In order to generate a forest in F_m uniformly at random, we first generate a number r in $\mathbb{Z}_{\#F_m}$ (called an *index*) uniformly at random. Then, we go through the process of whittling down r by successively discovering which count sequence to use for that forest, and which indices to use for each tree of that forest. (Such data collectively is called a *decomposition* of the index r.) After the decomposition is constructed, we recur on each tree size of the decomposition, noting that if the ith tree has c_i nodes, it can be viewed as a forest (of its subtrees) of $c_i - 1$ nodes, the root itself being one node. The recursion terminates when we are faced with generating a forest of one node with index zero, at which point we return a leaf.

Composing decompositions. For any forest of n nodes whose first tree can have as many as h nodes (called the *head size*), composing a decomposition is itself a recursive process which relies on three algorithms, PTCOORDS, LENREM, and RUNCOORDS, producing two vectors, *sizes* and *idxs*. PTCOORDS identifies which column of P_n that r is in (say it's the jth) and reduces the index and provides a new head. LENREM identifies which tower of L_{nj} that the reduced index is in (say it's the kth) and produces a re-reduced index, remainder index, and a remaining node count for subsequent trees. RUNCOORDS converts the re-reduced index into a sequence of indices for each of the k trees. We then recur on the remaining node count, the new head minus 1, and the remainder index, and we append a sequence of k entries of j to the front of the result's *sizes*, and also the sequence of indices to the front of the result's *idxs*.

This process is started with a call to DECOMP(m, m, r).

The three helpers are listed below.

LENREM sends two of its outputs to RUNCOORDS, which uses a binary search to

Algorithm 1 DECOMP

Require: A node count n, a head h, an index $r \in \mathbb{Z}_{\#F_n}$.

1: set *sizes* and *idxs* to be empty lists
2: **if** $n \leq 0$ or $h < 1$ **then**
3:    **return** $(sizes, idxs)$
4: **else if** $h > n$ **then**
5:    $h \leftarrow n$
6: **end if**
7: $(r', h') \leftarrow$ PTCOORDS(n, h, r)
8: $(k, n', r'', x) \leftarrow$ LENREM(n, h', r')
9: $frontidxs \leftarrow$ RUNCOORDS(h', r'', k)
10: set $frontsizes$ to be a list of k copies of h'
11: $(backsizes, backidxs) \leftarrow$ DECOMP$(n', h' - 1, x)$
12: **return** (append$(frontsizes, backsizes)$, append$(frontidxs, backidxs)$)

Algorithm 2 PTCOORDS

Require: A node count n, a head h, an index $r \in \mathbb{Z}_{\#F_n}$.

1: $r' \leftarrow r, h' \leftarrow h$
2: **while** $P_{nh'} \leq r'$ **do**
3:    $r' \leftarrow r' - P_{nh'}$
4:    $h' \leftarrow h' - 1$
5: **end while**
6: **return** (r', h')

Algorithm 3 LENREM

Require: A node count n, a new head h', a reduced index r'.

1: $k \leftarrow \lfloor n/h' \rfloor$
2: **while** $L_{nh'k} \leq r'$ **do**
3:    $r' \leftarrow r' - L_{nh'k}$
4:    $k \leftarrow k - 1$
5: **end while**
6: $n' \leftarrow n - (k \cdot h')$
7: $c \leftarrow \lfloor L_{nh'k} / R_{h'k} \rfloor$
8: $r'' \leftarrow \lfloor r'/c \rfloor$
9: $x \leftarrow r' \mod c$
10: **return** (k, n', r'', x)

determine what the indices should be for each of the k trees of nodesize h', based on the re-reduced index r''. This approach is needed since r'' is an index into one of the $\binom{\#T_{h'}+k-1}{k}$ nondecreasing sequences from seg_k to $seg_{\#T_{h'}}$, but this quantity is a sum (as per Theorem 5.1), so we have to figure out where r'' is in that sum without examining $\#T_{h'} - 2$ individual binomial coefficients, as $\#T_{h'}$ can get very large. (Indeed, this is the part that is significantly different from the uniform random generation algorithm mentioned below Assumption 5.2.)

Algorithm 4 RUNCOORDS

Require: A new head h', a re-reduced index r'', a length k.

1: set $idxs$ to be an empty list
2: $top \leftarrow R_{h'1}$, $prev \leftarrow 0$
3: **for** $t \in \{k, k-1, \ldots, 2\}$ by -1 **do**
4:     $i \leftarrow top + 1$, $j \leftarrow 1$, $mid \leftarrow \lfloor (i+j)/2 \rfloor$
5:     $total \leftarrow \binom{top+t-1}{t}$, $pen \leftarrow total - 1 - r''$
6:     $found \leftarrow$ false
7:     **while** not $found$ **do**
8:         $c_1 \leftarrow \binom{mid+t-1}{t} > pen$, $c_2 \leftarrow pen \geq \binom{mid+t-2}{t}$
9:         $found \leftarrow (c_1$ and $c_2)$
10:         **if** not $found$ **then**
11:             **if** c_1 **then**
12:                 $i \leftarrow mid$
13:             **else**
14:                 $j \leftarrow mid$
15:             **end if**
16:             $mid \leftarrow \lfloor (i+j)/2 \rfloor$
17:         **end if**
18:     **end while**
19:     $prev \leftarrow top - mid + prev$
20:     insert $prev$ onto the back of the list $idxs$
21:     $top \leftarrow mid$, $r'' \leftarrow r'' - (total - \binom{mid+t-1}{t})$
22: **end for**
23: insert $r'' + prev$ onto the back of the list $idxs$
24: **return** $idxs$

We remark in passing that building a rooted forest of m nodes corresponding to an index r takes slightly more than $O(m)$ time, but definitely within $O(m^2)$ time.

5.2.3 Experiments on Rooted Forests

Now that we have a reliable method to generate rooted forests uniformly at random, we wish to generate some statistics from them. We notice that in the first stage of an agreement phase, PET SNAKE will perform half of the total agreements necessary to ensure all symbols are pairwise agreed. In the next stage, it performs a quarter of them. And so on. Hence, at any point, about half of the deletions that can be performed during a stage will be performed on average.

Now, if a deletion gets performed, then that deletion's children will then be available to be deleted. Examining the consequences for the model, we see that only the roots of the trees in the forest are available for deletion, so when such a deletion is performed, the corresponding root must be eliminated. This, however, means that that root's children are now roots in the forest. This operation of deleting a root and promoting its children we call a *lift*.

Hence, about half of the roots are lifted in a given stage. (Such an action we will refer to as a *parallel lift*.) The agreement phase is not complete until all the nodes in the forest are eliminated. To get a handle on time estimates, it is pertinent to ask how many roots exist at a given time, and how many times to we expect to perform parallel lifts until the forest is eliminated. Since we do not have theoretical answers to these questions, we design an experiment as follows: for various $m \leq 1000$, we perform the Experimental Procedure (see Figure 5.1) several times (say, s times). Throughout each procedure run, we count the number of roots that the forest has (once before each parallel lift) so as to calculate the average when the forest is eliminated, and we also count the number of times we have to parallel lift.

Once the number of parallel lifts and the average number of roots are calculated, we do it again for the same forest. This is repeated s times. Once these s procedures

- Construct a rooted forest of m nodes uniformly at random via the above procedure.
- While it is nonempty,
  - take note of the number of roots of the forest,
  - uniformly at random choose half of the roots, and
  - lift them from the forest.
- Calculate the average number of roots the forest had.

Figure 5.1: Experimental Procedure

are complete, we choose another rooted forest of m nodes uniformly at random and perform the procedure again. We construct t such forests (each giving rise to s procedures), and a global average of number of parallel lifts required and number of roots appearing at any point are calculated.

This procedure was performed for $s = t = 1000$ and $m \in \{50, 100, 150, \ldots, 1000\}$ and the results are summarized in Table 5.2.

m	Avg parallel lifts	Avg roots	m	Avg parallel lifts	Avg roots
50	25.4741	3.9869	550	100.283	11.4762
100	38.7107	5.3268	600	105.187	11.9226
150	49.2455	6.330	650	109.466	12.4351
200	56.9224	7.3193	700	112.619	13.01
250	65.1864	7.9930	750	119.717	13.1435
300	71.7676	8.7246	800	123.128	13.6412
350	78.5635	9.3096	850	125.295	14.2402
400	83.6236	10.0201	900	129.423	14.5746
450	89.7707	10.4778	950	133.577	14.9439
500	94.8623	11.0328	1000	135.625	15.4812

Table 5.2: Experimental Procedure Results ($s = t = 1000$)

If we multiply the average roots by the average parallel lifts and plot this result for all twenty pairs, we discover that the plot forms a near-straight line of slope approximately $\frac{40}{19}$. This isn't too surprising, since in each parallel lift we eliminate

about half the roots, and the roots multiplied by the parallel lifts (if we eliminated every root per lift) should give us the total number of nodes in the forest. Further, if we multiply the number of roots by itself and plot this, we get a near-straight line of slope approximately $\frac{9}{38}$. From these two observations, we propose the following:

Heuristic 5.1. *A rooted forest of m nodes chosen uniformly at random will have, on average, $\sqrt{\frac{9}{38}m} \approx 0.4866\sqrt{m}$ roots on average as its corresponding set of deletions get deleted through an agreement phase.*

Further, the number of parallel lifts required to eliminate such a forest is on average approximately $\frac{40}{19}m/0.4866\sqrt{m} \approx 4.3264\sqrt{m}$.

Since this is based on experimental results only, we dare not call it a theorem. It will, however, be good enough for the work ahead.

5.3 The Transfer Formula

At long last, we are in position to calculate the time PET SNAKE takes given some data from a corresponding Clump Search software run using the PET SNAKE flow with threshold 20. We already have high-level time descriptions of each PET SNAKE phase from Section 4.1, and we also have low-level clock counts of each operation in Section 4.4. What we do here is predict how many of each operation will be performed through a particular run and put all the time costs together. At this time, we confess that an actual formula is not forthcoming, but the predictive mechanism is instead, predictably, a computer program.

Input data. After the software run is executed, it generates data for each turn it goes through. Each turn is broken up into segments and glue phases. The data we collect is (a) the dimensions of each symbol in the state before each segment's

execution, referred to as *sd*, and (b) the number of deletions that have occurred in each agreement phase in each segment's execution, referred to as *aops*, where $aops_{ij}$ is the number of deletions in the jth agreement phase of the ith segment. The most important thing about the symbol sizes is the size of the largest symbol, since that is often the factor that determines PET SNAKE's clock counts in most phases. We do not collect symbol sizes before each agreement phase since the size of the largest symbol is not expected to drop dramatically before a glue phase, and because collecting such data may become unwieldy very quickly. From this standpoint, our time estimates are more pessimistic than the actual times.

There are two main parts to computing an estimate: computing multipliers, and computing a turn estimate.

5.3.1 Multipliers

We have seen that PET SNAKE will go through a guess tree, but it will also split its MPUs according to how many symbols are in the current state. Inbetween splits, a subtree of the guess tree emerges which is fully evaluated. After another split, however, two subtrees from that node emerge, but we only count the time for one of them. Care must be taken to model this behavior accurately. For convenience, we create two 1-based arrays of multipliers to be visualized as going down the guess tree, one per depth of turns, both starting with 1 at the initial turn. The first array θ_1 is a local subtree multiplier, which increases by a power of two for each lower level, until it reaches a node where the board splits, at which point its next value is set to 1. The second array θ_2 is a cumulative multiplier: the next term is the same as the previous term, unless the first array reset itself to 1 in that term, in which case that term is the previous term of the first array multiplied by the previous term of itself. In symbols, for $n > 1$,

$$\theta_2(n) = \begin{cases} \theta_2(n-1) & \text{if } \theta_1(n) \neq 1 \\ \theta_2(n-1) \cdot \theta_1(n-1) & \text{if } \theta_1(n) = 1 \end{cases}$$

To calculate the turn cost for a collection of turns at the same depth, we use Assumption 5.1: we calculate the cost for the given turn and multiply it by both multipliers of that depth.

5.3.2 Turn Estimates

To calculate the cost of a turn, we calculate the cost of each segment and each glue phase. To calculate the cost of a segment, we use the sizes of all the symbols prior to that segment's execution, and from there calculate the cost of each agreement phase and extraction phase in that segment. In each phase there is massive parallelism, but a phase won't end until the last MPU finishes its operations for that phase; hence, we must find the dimensions of the largest symbol and use those as a basis for each phase.

We use TURNCOST, given below, to get a handle on the turn cost. This uses three helpers, AGREEP, GLUEP, and EXTRACTP, to calculate the cost of a whole agreement phase, glue phase, and extraction phase, respectively. We also build a function q which, when given a number of symbols, outputs the number of MPUs necessary to process those symbols (found in Table 5.1).

We turn our attention to the three helpers. For ease of computation, we note that the number of symbols for a full break of something like AES or PRESENT is less than 2049, so we suppose $g = 2$ throughout. (This also implies only one extraction stage per extraction phase.) Further, we suppose $\gamma = 0.5$ and $\gamma' = \frac{1}{8}\frac{(1-\gamma)(1+3\gamma)}{(1+\gamma)^2}$, as provided in the Linear Algebra Hardware analysis of Section 4.4. We also assume that the expected number of columns for any glue is $d = 2^{20}$, as discussed in the

Gluing Hardware analysis of Section 4.4.

Further, we construct a function DIMS that, when given a number of MPUs q, outputs the dimensions of the active area of q MPUs as PET SNAKE would lay them out; that is, $\text{DIMS}(q) = (q_1, q_2)$ such that $q_1 \cdot q_2 = q$, $q_1 \leq q_2$, and $q_1 = q_2$ or $2q_1 = q_2$.

For any symbols (A_i, L_i) with dimensions $w_i \times y, w_i \times c_i$ and (A_j, L_j) with dimensions $w_j \times y, w_j \times c_j$, we write a few low-level functions inspired by the hardware analyses in Section 4.4.

First, we construct a function MOVEC that, when given w_i, c_i, and number p of MPUs the symbol has to move through, outputs $2 \cdot p \cdot (w_i + c_i)$. This is because the columns of A and the rows of L each can grow up to 2047, so it will take at worst 2 clocks to move a row of A or a column of L.

Further, $\text{AGREEC}(w_i, w_j, y, c_i, c_j) = \lceil y + \gamma' \cdot (w_i + w_j) \rceil + 2(c_i + c_j)$.

Also, $\text{GLUEC}(w_i, w_j, y, c_i, c_j) = \text{AGREEC}(w_i, w_j, y, c_i, c_j) + 8c_i + 8d/6$, again with

---

Algorithm 5 TURNCOST

Require: sd and $aops$ for the turn under consideration.

1: $out \leftarrow 0$
2: **for** $i \in \{1, 2, \ldots,$ rows of $aops\}$ **do**
3: $\quad maxw \leftarrow$ maximum rowcount of a symbol in sd_i
4: $\quad maxc \leftarrow$ maximum column count of an L-part of a symbol in sd_i
5: $\quad y \leftarrow$ columns of an A-part of a symbol in sd_i // they're all equal
6: $\quad n \leftarrow$ number of symbols in sd_i
7: $\quad$ **for** $j \in \{1, 2, \ldots,$ entries in $aops_i\}$ **do**
8: $\quad\quad out \leftarrow out + \text{AGREEP}(maxw, y, maxc, q(n), aops_{ij}, n)$
9: $\quad$ **end for**
10: $\quad out \leftarrow out + (\text{EXTRACTP}(maxw, y, maxc, q(n)) \cdot$ entries in $aops_i)$
11: $\quad$ **if** $i <$ rows of $aops$ **then**
12: $\quad\quad out \leftarrow out + \text{GLUEP}(maxw, y, maxc, q(n))$ // no glue after last segment
13: $\quad$ **end if**
14: **end for**
15: **return** out

---

the supposition of worst-case behavior in the glue phase.

In addition, EXTRACTC$(w_i, c_i) = 18 \cdot 2^{11} + 13c_i + w_i$. These are just the raw costs of the agreement, glue, and extraction operations inside an MPU, but their respective phases involve some setup, moving symbol costs, and in one case, the probabilistic analysis above.

With these in place, it is not difficult to construct EXTRACTP.

Algorithm 6 EXTRACTP

Require: w, y, c, q as discussed above.

1: $out \leftarrow 2\cdot$EXTRACTC(w, c) // extract equations from g syms
2: $out \leftarrow out + 2w(\lfloor \log_2(q) \rfloor + 1)$
3: $out \leftarrow out + 2y(q-1)$ // total move cost of the A-parts
4: $out \leftarrow out + 2\cdot$AGREEC$(w, y, y, c, 1)$ // agree g syms with S_0
5: $out \leftarrow out + 2\cdot$GLUEC$(w, y, y, c, 1)$ // glue g syms with S_0
6: **return** out

Line 2 of EXTRACTP can be justified since as the equations continuously get extracted, more and more 1s appear in the row-reducer's `A` part, so we rapidly approach the cost of $2w$ clocks for that reduction, as per [6], and we pay the time cost for $\lfloor \log_2(q) \rfloor + 1$ such reductions.

It is also not difficult to describe GLUEP.

Algorithm 7 GLUEP

Require: w, y, c, q as discussed above.

1: $(q_1, q_2) \leftarrow$DIMS(q)
2: **return** GLUEC$(w, w, y, c, c)+$MOVEC$(w, c, q_1 + q_2)$

Let k be $\log_2(q)$. To calculate the agreement phase cost, we build two more functions $\eta(a, k)$ and $\mu(a, q)$. Now, given a total number of deletions m, we use Heuristic 5.1 to define a function ONDECK to calculate how many of those deletions we can actually perform (i.e., the roots of a forest F_m), and to define a function LIFTS to calculate how many times we expect to perform parallel lifts. For simplicity,

we denote ONDECK(m) by a (for *available* deletions). As discussed earlier, the agreement stage in which a deletion is performed is the same stage that about half of the a deletions will be performed. It then becomes critical to determine in which of the k stages that will be on average.

Recall that n is the number of symbols. Since half of the total $n-1$ agreement costs occur in the first stage, a quarter occur in the second stage, and so on, a given available deletion will have a $1/2$ chance of being performed in the first stage. Hence, for the a deletions, we will detect at least one of them in the first stage with probability $\frac{2^a-1}{2^a}$. If we don't detect any (probability $\frac{1}{2^a}$), we look at the second stage. But here, the story is the same: half of the agreement costs are left to pay (potentially), but half of those will be in the second stage. So, each of the a deletions has a $1/2$ chance of being performed, and so we will detect at least one of them with probability $\frac{1}{2^a} \cdot \frac{2^a-1}{2^a}$. And so on for the first $k-1$ stages. In the last stage, we must detect a deletion with probability $\frac{1}{2^{a(k-1)}}$.

Now, if we detect in the first stage, we would have to pay up to half of the total agreement costs (as per Section 4.1). If we detect in the second stage, we would have to pay up to three quarters of the total agreement costs: half for the first stage, and a quarter for the second. And so on. Thus, we define $\eta(a,k)$ as the fraction of the total agreement costs we have to pay, and we calculate it as

$$
\begin{aligned}
\eta(a,k) &= \overbrace{\frac{2^a-1}{2^a}\left(\frac{1}{2}\right)}^{\text{1st stage}} + \overbrace{\frac{1}{2^a}\frac{2^a-1}{2^a}\left(\frac{3}{4}\right)}^{\text{2nd stage}} + \overbrace{\frac{1}{2^a}\frac{1}{2^a}\frac{2^a-1}{2^a}\left(\frac{7}{8}\right)}^{\text{3rd stage}} + \cdots + \overbrace{\frac{1}{2^{a(k-1)}}(1)}^{k\text{th stage}} \\
&= \frac{1}{2^{a(k-1)}} + \sum_{i=1}^{k-1} \frac{2^a-1}{(2^a)^i} \cdot \frac{2^i-1}{2^i} \\
&= \frac{1}{2^{a(k-1)}} + (2^a-1)\sum_{i=1}^{k-1}\frac{1}{2^{ai}} + (1-2^a)\left[\left(\sum_{i=0}^{k-1}\frac{1}{2^{(a+1)i}}\right)-1\right] \\
&= \frac{1}{2^{a(k-1)}} + \frac{2^{ak+a}-2^a}{2^{ak}} + \frac{(1-2^a)2^{a+1}(2^{ak+k}-1)}{(2^{a+1}-1)2^{ak+k}} \\
&= \frac{1}{2^{a(k-1)}} + (2^{ak+k+a}-2^{k+2a+1}+2^{k+a}-2^{a+1}+2^{2a+1})/((2^{a+1}-1)(2^{ak+k})) \\
&= \frac{2^a}{2^{a+1}-1} + \frac{2^a-1}{(2^{a+1}-1)2^{(a+1)(k-1)}}.
\end{aligned}
$$

We remark in passing that since a computer will ultimately be performing the calculation, the closed form of η is not strictly necessary; the defining form can be computed in $O(k)$ time. Nonetheless, the closed form makes it plain that η is always over $\frac{1}{2}$, it is slightly over $\frac{1}{2}$ most of the time (corresponding to detecting a deletion in the first or second stages), and it gets closer to 1 as k gets closer to 1, achieving equality at $k=1$.

But agreements are more than their agreement costs; the symbols must also move to different snakes through different stages. We still use the same probability field to determine which stage an available deletion is detected, but we note that if a detection is in the first stage, the symbols do not move. If in the second, the symbols move through $\frac{2q_2}{2}-1$ MPUs. If in the third, they move through an additional $\frac{2q_1}{2}-1$ MPUs. If in the fourth, they move through an additional $\frac{2q_2}{4}-1$ MPUs. If in the fifth, they move through an additional $\frac{2q_1}{4}-1$. And so on. Hence, we define $\mu(a,q)$ as the total number of movement costs that must be paid while trying to detect a

deletion:

$$\mu(a,q) = \overbrace{\frac{2^a-1}{2^a}(0)}^{\text{1st stage}} + \overbrace{\frac{1}{2^a}\frac{2^a-1}{2^a}\left(\frac{2q_2}{2}-1\right)}^{\text{2nd stage}} + \overbrace{\frac{1}{2^a}\frac{1}{2^a}\frac{2^a-1}{2^a}\left(\frac{2q_2+2q_1}{2}-1\right)}^{\text{3rd stage}} + \dots$$
$$+ \underbrace{\frac{1}{2^{a(k-1)}}(h-1)}_{k\text{th stage}}$$

where

$$h = \begin{cases} \sum_{i=1}^{(k-1)/2} \frac{2q_2+2q_1}{2^i} & \text{if } k \text{ is odd} \\ \frac{2q_2}{2^{k/2}} + \sum_{i=1}^{(k-1-1)/2} \frac{2q_2+2q_1}{2^i} & \text{if } k \text{ is even} \end{cases}$$

which evaluates to

$$h = \begin{cases} 2(q_1+q_2)(1-\frac{1}{2}^{(k+1)/2}) & \text{if } k \text{ is odd} \\ 2(q_1+q_2)(1-\frac{1}{2}^{k/2}) + \frac{2q_2}{2^{k/2}} & \text{if } k \text{ is even.} \end{cases}$$

Since this can be calculated in $O(k)$ time, we do not need to find its closed form.

Since on average we expect to perform LIFTS(m) parallel lifts, we expect to return to the first agreement stage this many times during an agreement phase. Finally, after the last deletion is performed, we must go through all the stages one more time so that PET SNAKE is assured the system is in agreement, so this translates into paying the full $n-1$ agreement costs and the full $2(q_1+q_2-2)$ moves.

Algorithm 8 AGREEP

Require: w, y, c, q as discussed, number of deletions m, number of symbols n.

1: $(q_1, q_2) \leftarrow$DIMS(q), $k \leftarrow \log_2(q)$, $a \leftarrow$ONDECK(m)
2: $u \leftarrow ((\eta(a,k)\ \cdot\ \text{LIFTS}(m)) + 1)\ \cdot\ (n-1)\ \cdot\ \text{AGREEC}(w,w,y,c,c)$
3: $v \leftarrow$MOVEC$(w, c, (\mu(a,q)\ \cdot\ \text{LIFTS}(m)) + 2(q_1+q_2-2))$
4: **return** $u+v$

5.3.3 Evaluating Total Cost

Once the multipliers and the turn costs are calculated for every turn in a Clump Search software run, evaluating the total cost of PET SNAKE is straightforward:

Algorithm 9 TOTALCOST

Require: θ_1, θ_2, and a list of turn costs tc.

1: $out \leftarrow 0$
2: **for** $i \in \{1, 2, \ldots,$ entries in $tc\}$ **do**
3: $\quad out \leftarrow out + (\theta_1(i) \cdot \theta_2(i) \cdot tc_i)$
4: **end for**
5: **return** out

Note that TOTALCOST outputs clocks. Since PET SNAKE can run at 1 GHz, we simply divide the output of TOTALCOST by 10^9 to get the number of seconds required. Since this will calculate a full cost over the entire guess tree, we only need a fraction of this total cost to get the time for an actual key, depending on where that key is situated in $\{0, 1\}^e$, where e is the number of key variables as discussed in Section 3.1.2.

5.4 Some Examples

In each example below, the listed cipher was attacked using the PET SNAKE flow with Clump Search guessing at threshold 20.

5.4.1 AES

In Section 3.2.2, attacks were made on AES to get a handle on δ. Since these attacks used the PET SNAKE flow with Clump Search guessing at threshold 20, they were also suitable candidates to use the Transfer Formula (which is to say, to apply TURNCOST and TOTALCOST appropriately) supposing we have to traverse the entire guess tree. The results are summarized in Table 5.3.

Rounds	Time Estimate
3	572 641 271 198 465 353 652 543 years
4	960 251 453 705 422 259 760 162 years
5	3 032 259 107 403 426 131 298 634 years
6	4 334 274 240 537 784 797 951 228 years
7	5 836 535 502 648 494 352 466 589 years
8	15 197 068 416 307 157 725 269 672 years
9	18 822 062 696 908 351 440 301 350 years
10	22 869 947 121 743 980 307 889 196 years

Table 5.3: PET SNAKE Estimated Runtimes for AES, Varying Rounds

Should we ever find a way to reduce δ for these attacks (currently 108), we will see a corresponding drop in these runtimes. For the moment, AES remains safe from an MRHS perspective.

5.4.2 DESL

DESL was attacked on Pink (4-12 rounds) and on Aoba (16 rounds) with key choice 1 using Greedy Method gluing. The δ values and PET SNAKE estimates are provided in Table 5.4.

Rounds	δ	Time Estimate
4	0	1 second
6	34	8 years
8	36	88 years
10	36	192 years
12	40	5960 years
16	40	9738 years

Table 5.4: PET SNAKE Estimated Runtimes for DESL, Varying Rounds

It comes as no surprise that these δ values are perfectly in line with those in Table 3.6.

5.4.3 PRESENT

In Section 3.2.7, attacks were made on PRESENT to determine if the all-zero key was weak from an MRHS perspective. Since these attacks used the PET SNAKE flow with Clump Search guessing at threshold 20, they were also suitable candidates to use the Transfer Formula supposing we have to traverse the entire guess tree. The results are summarized in Table 5.5.

Indeed, one would be tempted to categorize PRESENT as a serious cipher from an MRHS perspective.

As an aside, we re-examine the effect of multiple pairs for four rounds. Since we saw some nice improvements in Tables 3.21 and 3.22, predicting PET SNAKE's running time in these scenarios seems a natural next step. The results are summarized in Table 5.6.

Rounds	Key Choice 0	Key Choice 1
3	58 days	26 days
4	4 years	6 years
5	11395 years	9919 years
6	83 398 556 years	60 249 285 years
8	1 344 365 235 years	1 718 089 488 years
10	54 614 576 831 years	110 123 041 028 years
12	113 252 409 157 years	17 837 698 389 years
14	321 746 362 627 years	760 502 302 951 years
16	22 584 022 980 years	34 697 669 929 years
18	65 313 686 391 years	41 011 552 683 years
20	65 618 679 316 years	108 605 027 465 years
22	125 820 322 016 years	196 741 697 231 years
24	164 936 979 378 years	197 304 216 914 years
26		258 049 383 864 years

Table 5.5: PET SNAKE Estimated Runtimes for PRESENT, Varying Rounds and Key Choice

Number of Pairs	Increment Way	Rotation Way
2	4 years	30 years
4	165 days	5 years
6	1 day	35 years
8	430 days	

Table 5.6: PET SNAKE Estimated Runtimes for 4 Rounds of PRESENT, Varying Pairs

5.4.4 KeeLoq

KeeLoq was attacked on Pink with key choice 1 using Greedy Method gluing. The δ values and PET SNAKE estimates are provided in Table 5.7.

Rounds	δ	Time Estimate
96	0	13 minutes
112	30	394 days
128	34	3730 years
160	36	4147 years
192	36	1 602 772 years
224	40	802 197 years
256	40	44 172 143 years

Table 5.7: PET SNAKE Estimated Runtimes for KeeLoq, Varying Rounds

Again, these δ values are consistent with those in Table 3.8.

5.4.5 KATAN

KATAN was (again) attacked on Pink using a key choice of 1 and the Greedy Method of gluing. The δ values and PET SNAKE estimates are provided in Table 5.8.

Rounds	δ	Time Estimate
40	28	1 hour
60	48	1900 years
70	58	11 651 327 years
80	61	217 594 493 years
100	63	2 627 195 600 years
128	72	2 975 185 659 042 years

Table 5.8: PET SNAKE Estimated Runtimes for KATAN, Varying Rounds

Given this, KATAN too seems safe from an MRHS perspective.

Chapter 6

Conclusion

6.1 The Present

We hope to have provided a definitive work on the capabilities of MRHS, illustrating what is currently within reach of this algorithm, and some ideas as to how to improve its execution. We also hope to have provided implementations that are completely *modular*: not only was the software designed so as to be as abstract and as flexible as possible to incorporate new flows, guessing strategies, gluing strategies, ciphers, thresholds, and metrics collections, but PET SNAKE itself is an inexpensive modular design allowing for $2^{2\lambda}$ MPUs to be used for any $\lambda \in \mathbb{N}$. Further, if a particular method of row-reduction or multiplication is preferred to what is currently designed, the framework is there to just replace those parts. In addition, our Transfer Formula calculations were also designed to be modular: should better methods of estimation become available, the affected aspects of the calculation can be replaced without breaking the whole process. Even if one judges the end results of MRHS to not be particularly compelling over existing algebraic attacks like SAT solvers or Gröbner bases, the modular design characteristics described ought to be pleasing to any implementor.

What is interesting about MRHS is that guessing can be mitigated if one uses larger amounts of memory, though the algorithm will run (by lowering threshold) even if it doesn't have large memory at its disposal. Another nice aspect of MRHS is that, unlike SAT solvers or Gröbner bases, the attackers get some kind of feedback as to where the attack is and roughly how much longer it might take, provided that they perform an Abbreviated Search first before a full search. Discovering δ is key for these predictions.

We have seen that, in large part, most serious ciphers are relatively safe from an MRHS-only approach, even if we use PET SNAKE on them. Indeed, DESL's expected runtimes compare unfavorably to the nine days COPACOBANA takes for a brute-force approach on a full DES. On the other hand, smaller-round variants of many ciphers become within reach, especially PRESENT and KATAN, which cannot currently be attacked with COPACOBANA owing to their larger key sizes, and in some cases, an MRHS attack is definitely less expensive than a full brute-force software search. Also, we have discovered that MRHS is sensitive to some options such as the gluing and guessing variations, and in at least one case, using a simpler flow actually reduces overall runtime in software (and hence hardware).

Further, the use of multiple plaintext/ciphertext pairs gives little encouraging news for AES and DESL, but some encouraging news for reduced-round versions of PRESENT, KeeLoq, and KATAN, and possibly other ciphers which have keysizes larger than the plaintext size. In some cases, increased rounds will reduce the effectiveness of multiple pairs, but in other cases it is unclear if such a reduction will be seen.

Finally, different parts of this work might be of independent interest. For example, we implement generating rooted trees uniformly at random without much cost in either memory or time. Further, the JONES architecture might have play in

other environments where sparse-matrix operations are prevalent, and the hardware matrix multiplier can also be used elsewhere.

6.2 The Future

To develop this work further, one can pursue several avenues. For example, we focus exclusively on MRHS variants using pairwise agreeing via the Agreeing1 Algorithm. Other agreeing processes are available, but were not chosen owing to their increased use of memory. Such processes can use less time than Agreeing1, however, so they are certainly worth exploring in both a hardware and software context, perhaps when larger-memory machines are available.

Other potential avenues are discussed below.

6.2.1 Statistics

As it was mentioned earlier, the necessary theoretical tools to predict the number of agreement phases or deletions over arbitrary cryptosystems are not currently available, but such things would certainly add value to this algorithm.

Further, performing a full analysis to validate Assumption 5.1 would also be valuable.

6.2.2 Software

One option would be to modify the software to perform realtime exacting memory calculations (both memory consumed by the state and memory required to perform agreement, extraction, and gluing) and then provide an option to glue symbols up to the actual available memory instead of to some arbitrary threshold.

Another option is to simply run the software on even larger memory systems. At the time of this writing, 1TB memory systems are in modern production; perhaps a nice drop in δ can be discovered for the ciphers examined. Certainly such attacks would make it easier to explore the effectiveness of increased pairs and increased rounds.

Further, one can reimplement to take advantage of multi-core processors. This would induce a near-linear speedup in agreeing, equation extraction, and gluing, and (in this author's view) would be preferable to trying to split an MRHS attack across many different computers, since the communication costs between them would be prohibitive using standard 10/100 or even 10/1000 Ethernet capabilities.

6.2.3 Hardware

Other options are hardware based. For example, the current specification of PET SNAKE relies on 45nm fab technology, but 32nm fab processes are on the horizon (and already here in the case of memory chips). Simply making everything using this newer technology would, in effect, double the amount of gates and components usable on each chip. One could then do a feasibility study to see where it would be best to allocate those resources.

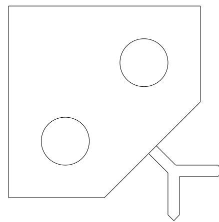
One can also slightly modify the MCP so that 2-symbol states can be processed on one MPU. Indeed, such a change can already be theorized now, but the associated Transfer Formula would have to change.

Another thing to note is that PET SNAKE keeps $g = 2$ as much as possible, but this means that a significant portion of the traffic controller is not used. Perhaps it would be of benefit to have PET SNAKE keep $g = 4$ as much as possible, effectively allowing four-way splits instead of two-way splits, but splitting less often. Much of the timing analysis and the Transfer Formula would have to change to accommodate

this new arrangement.

Also, PET SNAKE is inherently limited to systems of 2047 variables, but the equation extraction process implies that we could have a way to attack systems of 4095 variables or larger by multiplying and row-reducing in creative ways using the existing components, with the associated time tradeoff.

Finally, all of PET SNAKE's estimated running times were made with only one version of its MCP in mind. There's nothing to prevent us from changing the MCP in such a way to tailor the MRHS algorithm to more favorable behavior; for example, we could 'swap it out' for an MCP implementing a Basic flow if we were attacking twelve rounds of PRESENT, or for an MCP implementing First Available Gluing if we were attacking a full DESL. (Indeed, such a swap would cut down PET SNAKE's running time by almost a factor of 2 in this case, but the full development of this we postpone.) Certainly other Transfer Formulas could be developed with these options in mind, given the existing algorithms.



Appendix A

Traffic Controller

Architecture

Each of the four chips has 256 pins in each cardinal direction, plus 512 pins for the MPU bus. This means that, together, buses between MPUs are 1024 wide, but the MPU bus is 2048 wide. Hence, each of the four chips is responsible for a quarter of each symbol the MPU stores. In addition, there are 56 pins in and 56 pins out to pass data across the MCP bus. It is through this bus that the MCP will direct the actions of the symbols and the MPU.

Each chip has twelve DRAM areas. Six are considered the 'active' DRAMs: one for receiving a new symbol in the motile snake, one for sending a symbol in the motile snake, three for storing the currently-fixed $g-1$ symbols, and one for storing the gather symbol and equation set. The first five DRAMs are 65 MByte each, enough to store a quarter of a full symbol; the sixth is 1 MByte. The other six are the 'passive' DRAMs used to store a state before a guess is committed. They are connected to their corresponding 'active' DRAMs. The first five of these are 234 MByte (enough for 18 quarters of symbols when combined) and the sixth is 18 MByte. In addition, for each of the five large active DRAMs there is a 1024×1024 flip-flop grid with two decoders used to record deletions. Finally, there is some control logic to handle

MCP commands.

Just in Time Process

These chips operate on a 'just in time' basis during the agreement phase, specifically during last agreement that must be performed before the snake must move. Label the two symbols that must be agreed S and T, where T is part of the motile snake, and suppose they are of roughly equal size. The MPU will process T second, that is, T provides the A_j and L_j in the MPU data flow description.

Just after the last deletion for L_j (if any) is sent from the hash table, the traffic controller begins to transmit A_j and L_j (taking care not to send the columns marked for deletion) to the next MPU in the path. During this time, it is receiving the A and L matrices from the next symbol in this snake; label them A_y and L_y. Also during this time, the MPU is also processing the columns from L_i the second time, getting deletions for that symbol. These three actions are done in parallel since they are using different buses and different active DRAMs. By the time L_i has gotten all its marks for deletion, at a minimum only half of the data for the incoming/outgoing symbols will be received/sent, since the transmission buses are only 1024 wide in either direction. Since A-parts get transmitted before L-parts, A_y will be received.

Now, the traffic controller will continue receiving/sending symbol data while it sends the A-part of a fixed snake (call it A_x; if $g = 2$, $x = i$) to the row reducer to begin agreement with the incoming symbol. Once done, A_y will be sent to the row reducer (and transmissions of symbols will temporarily be halted). Then the row reducer will do its processing (at which point transmissions of symbols will resume), followed by L_x being sent to the multiplier. By the time L_x is done being processed (the first time), L_y will be completely received, and so L_y is now sent to the multiplier for processing.

In situations where the symbols have mismatching sizes, there must naturally be halts in processing until the data is received. Time calculations are performed supposing all symbols are full maximal size, so actual performance can be better than these calculations.

Deletion Handling

Each active DRAM has attached to it a flip-flop grid to register deletions. When a column index to be deleted is received on the MPU bus, this grid will handle it, leaving the DRAM to continue sending columns. The handling works as follows: each flip-flop's value is determined by a different AND gate. The most significant 10 bits of the received index are decoded into 1024 horizontal lines out. Each such line attaches to a row of 1024 of these AND gates. The least significant 10 bits of the index are also decoded into 1024 vertical lines out. Each such line attaches to a column of 1024 of these AND gates. Hence, each index corresponds to exactly one flip-flop, and the received index's flip-flop will be set to 1.

When it is time to transmit the L-part of a symbol, the flip-flop grid shifts its values in the top row (cyclically) over so that there is a 0 or 1 corresponding to the column to be sent. If the upper-right flip-flop has 1, the column is not sent; else, it is. After the send (or no-send), the row shifts again. After 1024 shifts, the whole grid performs a cyclical shift up.

Note that the two grids for motile snake symbols are reset before the new symbol is agreed, but the grids for stationary snakes are not. Hence, their grids will cumulatively store deletion marks until it is time for them to move for the next agreement stage. In addition their values will be used in determining if a column will be sent to the multiplier when it is called for in future agreements in the current stage. (This doesn't impact timing concerns, but it does make the hash table's life

easier.) After a full agreement phase has been completed, all symbols are moved one MPU so that deletions are incorporated into DRAM. Of course, if in the last agreement stage a deletion has occurred, the whole agreement phase starts over.

Equation Extraction

During an equation propagation phase, the sixth DRAM is used to store incoming equations during mass row reductions. It also stores and receives the current equation set, which takes no more than $\frac{1}{2}$ MByte. It also receives the gather symbol (also no more than $\frac{1}{2}$ MByte) and sends it on.

Area Calculation

Each chip has $(((65 + 234) \times 5) + 19)$ MByte of DRAM, yielding 1514 MByte, or 3.18 cm^2. In addition there are 5×2^{20} flip-flops, for 0.22 cm^2 (since we manufacture these chips using the 45 nm DRAM process). Each decoder can be realized with two 3-8 decoders (outs inverted), one 4-16 active low decoder (outs inverted), and 1024 3-input NOR gates (with latches at the inputs so as to form a two-stage pipeline) for a total of < 0.001 cm^2 for all decoders. The control logic will not grow above 0.45 cm^2, and a complete chip of the traffic controller thus fits onto 3.9 cm^2.

Appendix B

Row Reducer

Architecture of `A`

The `A`-part's main workhorse is 4096×2048 JONES elements connected in roughly the same way that SMITH elements usually are. Readers familiar with SMITH in [5] will undoubtedly recognize the similarities; indeed, JONES is meant as an improvement to SMITH to handle sparse matrices. This improvement is helpful because in the early and middle stages of processing, many symbols' A-parts are sparse, and the vertical concatenation of such things will still be sparse. The connections between JONES elements are essentially those of SMITH, taking care to send the `OUT3` signal to the leftward element's `IN3` input, wrapping cyclically around.

We see from Figure B.1 that a JONES element is similar to a SMITH element, but there are three additional lines added: `IN3`, `OUT3`, and `live_col`. A row of 2048 flip-flops called the *LC row* will be set so that a given flip-flop holds 1 only when there is a 1 in that column of the JONES matrix, and 0 otherwise. (`U` does not require these flip-flops; `live_col` will be set in a different manner in them.) The leading LC flip-flop, along with some control logic, will be attached to every element's `live_col` signal. If there is a 1 in the leading column, `live_col` will be 1 for the entire array, and JONES processes its data exactly as SMITH does.

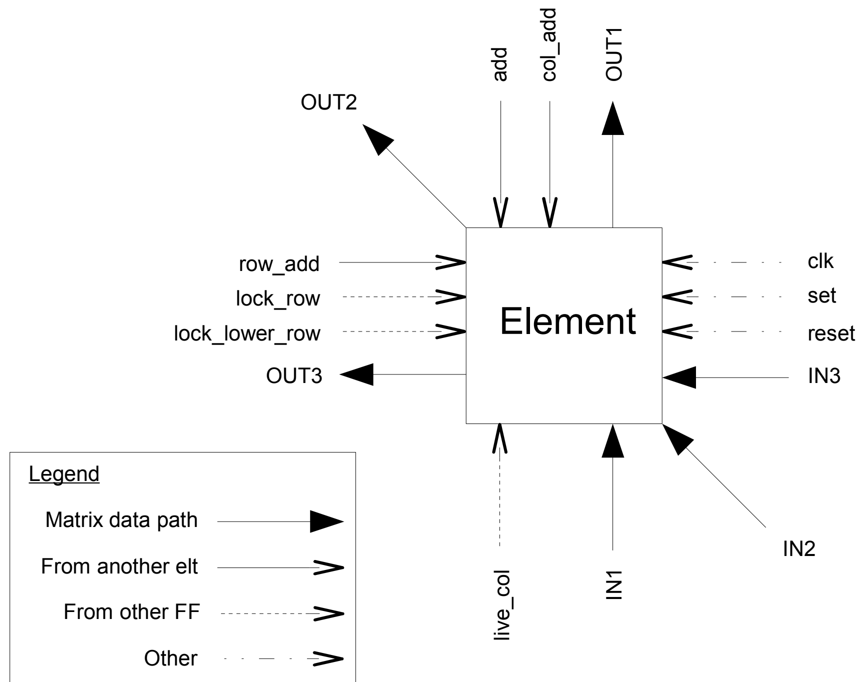


Figure B.1: JONES Element (High Level)

However, if there is not a 1 in the leading column, `live_col` is 0, and we cyclically shift the entire matrix leftward.

To bring about this change in behavior, the circuitry for SMITH was modified. The logical changes can be seen in Figure B.2. The gate depths for each path leading to the flip-flop are shorter than those in SMITH, so JONES can withstand faster clocking rates.

Turning our attention to some helper control circuits, we remark that `A`'s LC row will be initialized after both A_i and A_j are done being loaded into the elements in the following way: the LC row will be initialized to all zeroes, but OR gates will hang on them and the input lines (one per flip-flop). As A_i and A_j are loading, the LC row flip-flop for that column will constantly be updated with the cumulative OR of the current column value and the current flip-flop value. Hence, when the

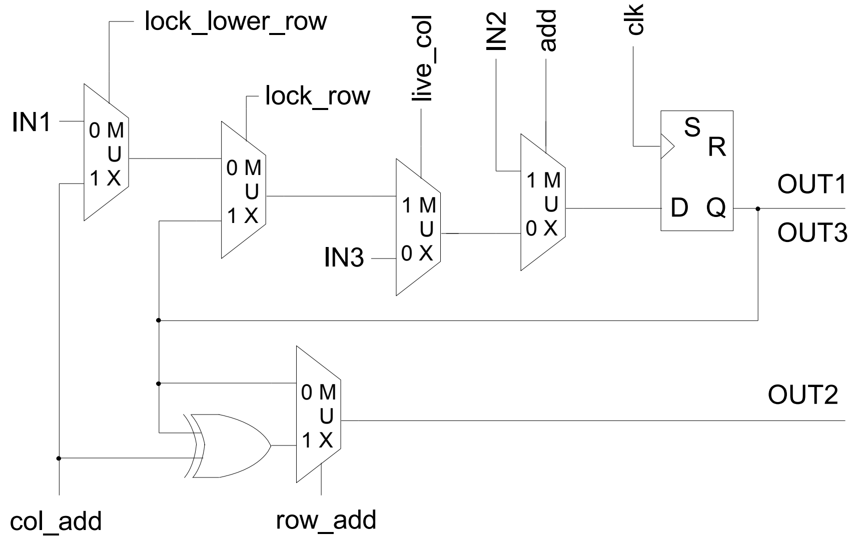


Figure B.2: JONES Element (Low Level)

data load is complete and before processing begins, the LC row will be populated as described earlier. The LC row will also be wired to cyclically left shift so that, during JONES processing, such a cyclic left shift will occur upon any add or left-shift in the JONES elements.

`A` also uses a column of 4097 flip-flops called the *LR column*, which will feed the `lock_row` and `lock_lower_row` inputs. When a new matrix is to be row-reduced, this column is initialized to all zeroes, with a 1 in the bottom position. A given flip-flop will feed its value to all `lock_lower_row` inputs in its row, and it will feed its value to all `lock_row` inputs in the row below it. As the elements perform their processing, when they detect that an add must be performed, the LR column will shift upwards, inserting a 1 in the bottommost position.

To facilitate equation extraction, `A` also uses a column of 2048 flip-flops called the *ZD column* connected so that the contents move cyclically upwards, a row of

2047 flip-flops called the *OD row* connected so that the contents move cyclically leftwards, and a flip-flop called the *OD flag*. The left 2047 JONES units of the top row of `A`, together with the top flip-flop of ZD, feed a tree of OR gates (with interstitial latches) called the *ZD tree*. This terminates in a single flip-flop which is the cumulative OR of these 2048 bits. (For a more detailed discussion of such a tree, see Appendix C, which discusses the multiplier.)

Also, each of the left 2047 JONES units of the top row of `A` will feed an XOR, whose other input is a corresponding element of the OD row. The output is fed back into the OD row. Finally, a row of 2048 flip-flops called the *GS row* is connected so the contents shift leftwards. Its last flip-flop is fed by the top row of `A`'s 2048th element.

Architecture of `U`

The `U`-part's main workhorse is 4096×4096 JONES elements, connected in roughly the same way that the `A`-part's are. The data lines `IN1`, `IN2`, `IN3`, and their `OUT` counterparts (collectively referred to as the *matrix data lines*) will be connected in the usual way so that the matrix data will follow a closed path at all times, but the other lines will be connected differently. Since the data in `A` determines which operations to perform on the matrix, we must direct the `U` elements to perform the same operations `A` does. To this end, every `add` input in `U` will be connected to `A`'s upper left `OUT1` signal. Every `row_add` input in `U` will be connected to `A`'s leftmost `OUT1` signal in the corresponding row. Every `live_col` signal in `U` will be connected to the LC row's leading flip-flop (the same one `A` uses). Every `lock_row` and `lock_lower_row` input in `U` will be connected to the LR column in the same way `A`'s are.

Before row-reducing a matrix, `U`'s JONES elements will be initialized to the

```
while (locks + overs < cols(A))
    if (live_col = 1)
        set z to 0
        while (z < rows(A) − locks and 0 is in upper left element of A)
            shiftup (both A and U)
        end while
        if (1 is in upper left element of A)
            add (both A and U)
        else
            shiftover (both A and U)
        end if
    else
        shiftover (both A and U)
    end if
end while
```

Figure B.3: Processing Algorithm

identity matrix. This can be accomplished by using a row of 4096 flip-flops (called the *identity row*), initialized with all zeroes except for a 1 in the leading column. During a clock cycle, the identity row stuffs its values in the JONES top row elements, and then on the next clock, the identity row cyclically shifts to the right as the JONES elements shift upward.

Process

When A_i comes across the MPU bus, `A` will stuff its JONES top row elements with these values while cumulative-ORing them with the LC row (which was initialized to all zeroes). Simultaneously, `U` will begin stuffing itself with the identity matrix. This continues when A_j comes across the MPU bus. When this is complete, row reduction can begin.

Since we have introduced a new operation (the left-shift), in Figure B.3 we detail the processing algorithm, which is realized with the JONES matrix as well as some additional control logic attached to the non-matrix-data inputs. We write

two quantities, *locks* and *overs*, which represent the number of adds and left-shifts, respectively, that have occurred in `A`. (The same counts will apply to `U`.) One may realize their sum with actual counters, or with a long row of flip-flops with all zeroes save for one 1. We also have a counter z for zeroes found in the current leading column of the `A` matrix.

After completion of the while loop in Figure B.3, `A` is finished, and the resulting r all-zero rows (if any) will appear on the top. `U` is not finished, however, so we must rotate it into proper position as shown in Figure B.4.

```
while ((locks + overs)  mod cols(U) ≠ 0)
       shiftover (U)
end while
```

Figure B.4: Rotating `U` into Proper Position

At this point, `U` is in proper position, and the r rows on top, when multiplied with the original values in `A`, will produce zero rows.

Remark. Even though `live_col` may be 1, the 1s that were in this column beforehand may have been eliminated by a previous add, and so a 1 left in this column would belong to the row responsible for eliminating the others; such a 1 would be in a locked row, where it will do us no good. Hence, z is still necessary. Life may be made substantially easier by using a tree of ORs (along with a column of ANDs and inverters attached to the LR column and the leading column's `OUT1` signals) attached to the leading column's `OUT1` signals to compute an accurate, real-time value for `live_cols`, but since there are 4096 rows, such a tree would require a column of interstitial flip-flops in the middle, which would impose a factor two slowdown in the linear algebra. With such a choice, z would no longer be necessary, and the first while loop above can be simplified. Further, the LC row would not be

needed.

We now turn our attention to sending U's data to the multiplier. Determining that the top row of A is a zero row is fairly easy: a tree of ORs branching from the 2048 JONES top row elements of A (with the required interstitial flip-flops in the middle to alleviate gate depth concerns) can be used. This means that data can only be sent every other clock cycle. In the event that there are no zero rows to begin with, a signal is sent decreeing that the two symbols this MPU is processing are agreed, and no further work needs to be done. Otherwise, row data (detailed below) is sent across the MPU bus to the multiplier until A no longer has a zero row on top. Then (if gluing) row data will be sent to the multiplier again, but the multiplier will put this data in a different place.

We determine which data needs to be sent at which time by observing the following. The MRHS algorithm calls for the multiplications $U\binom{L_i}{0} = UT_{ij}$ and $U\binom{0}{L_j} = UT_{ji}$, which produces matrices whose columns' bottom r values we must later compare. Because of the positions of the zero blocks, we only need the left rows(L_i) columns of U to produce Pr_{ij}, and the right rows(L_j) columns of U to produce Pr_{ji}. If we rotated the rows U cyclically downward r times, we would have the current contents of U. Since the top r rows in U contain the data we are immediately interested in, for r iterations we send the first rows(L_i) entries in the top row across the MPU bus and shift U and A up. If we are gluing, for rows(L_i) + rows(L_j) − r iterations, we again send the first rows(L_i) entries in the top row across the MPU bus and shift U up. (Otherwise we just perform the shiftups without sending this data.) We remark that, since A has only 2048 columns, the maximum rank of A is 2048, and so rows(L_i) + rows(L_j) − r must be less than or equal to 2048.

While the multiplier and hashtable are working to compute parts of UT_{ij}, U and

`A` will perform the necessary shiftups to get them back to their original positions, and then `U` will perform rows(L_i) shiftovers so that it is in position to send the data needed to compute parts of UT_{ji}. Data will be sent in a similar way once it is time to begin computing UT_{ji}.

Extraction Stage Process

Row reductions during an extraction process are done similarly as during agreeing or gluing. At the beginning of the extraction process, the ZD column and the OD row are reset to 0, and the OD flag is set to 1. At the end of each iteration in step 4 of the extraction process (see Figure 4.6), the `A`-part will be loaded with a chunk (that is, a group of 2^{11} columns) of UL, and the zero and one detection processes commence: the resulting bit in the end flip-flop of the ZD tree is moved into the corresponding position of the ZD column as this column (and the rows of `A`) are shifted cyclically upward once. This is repeated for every row in `A`. Note that four clocks of latency will be required in the beginning, as this process is pipelined. Once this is done, the ZD column will contain a 0 for every row of UL which (so far) has been all zero, and a 1 otherwise.

As the data from each row of `A` moves upwards for zero detection, it is cumulatively XORed in the OD row. Once all rows have been examined, the OD row is cyclically shifted left. The leftmost element of the OD row is connected to an AND gate with the OD flag, and the result of the AND is fed back into the OD flag. After all left shifts are complete, the OD flag denotes whether the result of (so far) adding the rows of UL will be an all-1 row. The OD row is then reset to 0.

Once all chunks of UL are processed, the OD flag will dictate if a nonhomogeneous equation exists, and the ZD column will dictate which rows, if any, correspond

to homogeneous equations. Once the extraction process finishes step 10 (see Figure 4.6), `A` will contain the matrix UA. The rows of `A` (and the ZD column) are rotated cyclically upwards once more: if the topmost entry in the ZD column is 1 and the OD flag is 1, the OD row will cumulatively XOR itself with that row. If not, OD is unchanged (from the all zero row). However, if the topmost ZD entry is 1 and the OD flag is 0, this row will not reappear at the bottom of `A`; instead, an all zero row will appear. If the topmost ZD entry is 0, the row will reappear at the bottom of `A`. Finally, the contents of OD are rotated into the bottom of `A`, where the 2048th element is set to the OD flag, and the contents of `A` are rotated up again an additional 4095 times. Thus, the nonhomogeneous equation will be in the top row of `A` (if it exists; else it is an all zero row), followed by any homogeneous equations. Then the ZD column is reset to all zeroes.

To convert the current equation set to the gather symbol, first the contents of `A` are rotated upwards once again, with the first 2047 elements being put onto the MPU bus and captured by the sixth active DRAM. The 2048th bit sent is 0 in this process. As each row rotates, the rightmost element of the GS row is populated with the top row of `A`'s 2048th element, and the contents of GS move leftward. After the first 2048 rotations, the rows have already been sent to the DRAM, and so the contents of the GS row are now sent to the sixth active DRAM, effectively forming the one column of the L-part of the gather symbol.

Resolution Stage Process

During the resolution stage, after S_0 and the gather equations have been row reduced, the ZD column is once again populated as the rows of `A` are rotated cyclically upwards. During this cyclic rotation, the top row is checked for equality with a row of 2047 zeroes and 1 one at the end. This is accomplished with an AND tree with

interstitial latches, done in a pipeline fashion. Sitting at the end of this tree is a flip-flop which cumulatively ORs itself as each row is rotated through. If this flip-flop is a 1 at the end of the rotation, the MPU signals the MCP that an inconsistency has been found. If no inconsistency is found, the ZD column is counted to check for maximal rank. If maximal rank has been achieved, the MCP is signaled. Otherwise the contents of `A` are sent across the MPU bus and stored in the sixth active DRAM of the traffic controller.

Area Calculation

Figure B.2 is provided so that the logic is easy to follow. Five multiplexers, an XOR gate, and a flip-flop yield 48 transistors per element. Since `A` has $2^{12} \times 2^{11}$ elements, we use Table F.2 and the 45 nm logic process to obtain an area of roughly 1.2 cm^2 for the elements. Since we only have a few more groups of 2048 flip-flops, XORs, ORs with some associated logic, the area of `A` does not expand much from this value. The area of `U`, however, is roughly twice of that of `A`, say 2.4 cm^2, since it has twice the number of JONES elements that `A` does. Hence, `A` and `U` together really fit on a chip.

Appendix C

Multiplier

Architecture

`Ur` and `Us` are two grids of flip-flops with some nontrivial logic attached. To make the process of comparing Pr_{ij} and Pr_{ji} as fast as possible, we do not compute these full matrices. Instead, we compute the first $r_{max} = 135$ rows of these matrices and ignore the rest. Hence, `Ur` is comprised of a grid of $2048 \times r_{max}$ flip-flops. `Us` however must be 2048×2048 flip-flops. They have otherwise identical architectures.

Figure C.1 illustrates the following description. Each flip-flop has an AND gate hanging off of it. All the AND gates in a given row will take their second input from that row's corresponding wire in the L bus. Then, each column of AND gates sprouts an XOR tree off of the AND outputs; this is to add these bits. (Successive pairs of AND outputs in a column are XORed together. Successive pairs of these XOR outputs in the column are again XORed. And so on until we get down to one XOR.)

Remark. To make transistor counts smaller, we observe that in reality, we could use NANDs instead of ANDs since their results are all XORed together, and XOR produces the same result if both inputs are inverted.

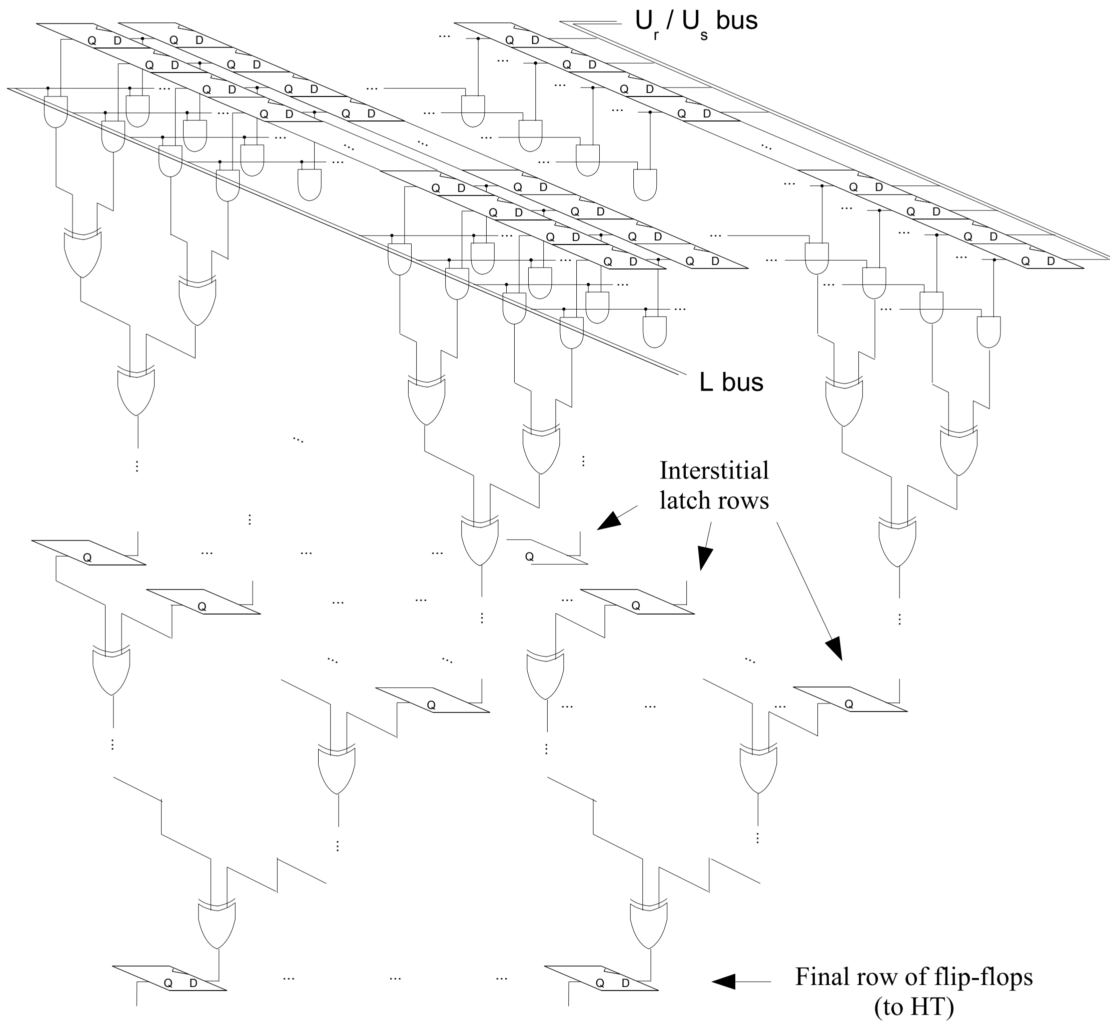


Figure C.1: `Ur` and `Us` (Low Level)

Since there are 2048 rows in `Ur` (and `Us`), the signal from a flip-flop would have to travel through 12 gates. This gate depth can induce a slower clocking rate, so we insert interstitial latches after the signals have traveled through 4 gates. This means that 256 such latches are inserted after 4 gates, and another 16 are inserted after another 4 gates, per column. Then the signal can travel through the remaining parts of the XOR tree (4 more gates) until it gets to the *final row* of flip-flops. This arrangement forms a small pipeline. The data from this final row will be sent to the

hash table. Since the hash table can process this row every clock cycle, we sent L_i and L_j columns through on each clock, and the interstitial latches merely induce a two-clock latency.

To facilitate equation extraction, it will be necessary to calculate the transpose of the A-part of a symbol, which is done in step 7 (see Figure 4.6). To accomplish this, an identity row of 2048 flip-flops (similar to that found in the `U`-part of `A/U`) is placed so that it can feed data to the L bus; a multiplexer will allow the MCP to choose whether the L bus is populated with this identity row.

Process

When (the left cols(L_i) components of) one of the top r rows of U gets sent from `A/U` across the MPU bus, it is picked up and stored vertically in the rightmost `Ur` flip-flop column. Then all the `Ur` flip-flops get shifted leftward. This continues for each of the r rows, up through r_{max} rows. If more than r_{max} rows are sent, they are ignored by `M/HT`. If gluing, then $\text{rows}(L_i) + \text{rows}(L_j) - r$ more rows of data are sent by `A/U`, and the multiplier picks these up and stores them vertically (in a similar fashion) in `Us`.

Then, the columns of L_i will be sent. For each such column, the multiplier will pick it up and put in on the L bus, where it will be multiplied to the contents of both `Ur` (and `Us`, simultaneously, if gluing). This method of multiplication mimics the *Method of Four Russians* technique (illustrated in [1]), only without the T-storage component. The result on the final flip-flops of `Ur` is a column of Pr_{ij}, which then gets sent to the hash table. If gluing, the result on the final flip-flops of `Us` is a column of UT_{ij} without its last r components; this gets sent to the adder across the MPU bus for storage. Note that, because of this use of the MPU bus, gluing will take at least twice as long as agreeing (possibly longer due to the hashtable).

After the columns of L_i are processed, A/U will send (the right $\text{cols}(L_j)$ components of) the top r rows of U, which is followed by, if gluing, $\text{rows}(L_i)+\text{rows}(L_j)-r$ more rows of data. Ur and Us will be flushed and reloaded with this data in the same way as before.

Then the columns of L_j will be sent, and they will be multiplied to the contents of Ur (and if gluing, Us) in the same way as before. The contents of the final flip-flops of Ur will once again be sent to the hash table, and the contents of the final flip-flops of Us will once again be sent across the MPU bus to the adder (though the adder will use them differently this time).

The multiplier is thoroughly used in an extraction stage, but only the flip-flops in Us are used. Different steps of the process (see Figure 4.6) will involve Us receiving data from either traffic control (columns from a symbol's L-part, or rows of a symbol's A-part) or from the row reducer.

Then, data will appear on the L bus, either from the left 2^{11} bits of the top column of the row reducer's U-part (after which the U-part is rotated cyclically upwards), or from the identity row. In either case, the multiplier will calculate a row of some multiplication matrix and send that back across the MPU bus to the row reducer's A-part. Then, more data will appear on the L bus, and another row is sent back. This is repeated until the row reducer's U-part, or the identity row, is exhausted.

Area Calculation

We count 2049 flip-flops per column, plus 272 interstitial latches, plus 2047 XORs, plus 2048 NANDs. Conservatively counting 10 transistors per XOR gate (which may happen in dense tree arrangements), this gives us $2049\cdot 12+272\cdot 6+2047\cdot 10+2048\cdot 4 = 54\,882$ transistors per column. Since there are $2048+r_{max} = 2183$ such columns, we

use the 45 nm DRAM process to get an area of 0.43 cm^2 for `Ur` and `Us` combined, including the amplification of the data on the long buses. (Since the multiplier shares a chip with the memory intensive hash table, we elect to use the DRAM process for `M/HT`.)

Appendix D

Hash Table

Architecture

The hash table internally splits into two identical separate subtables HT_0 and HT_1. Each HT_μ ($\mu = 0, 1$) contains $n_{DRAM} = 256$ DRAM blocks DB_ν ($\nu = 0, \ldots, n_{DRAM} - 1$), and is designed to process one (write or look-up) query per clock cycle. Elements to be stored or looked up in the hash table have a size of $r_{max} = 135$ bit, and each of the two subtables HT_μ has to store up to $n_{entries} = 2^{20}$ entries of $r_{max} + \log_2(n_{entries}) = 135 + 20$ bit each. To ensure a reliable collision resolution, each DRAM block will offer space for $4 \cdot n_{entries}/n_{DRAM} = 2^{14}$ table entries.

The hash table is used in both PET SNAKE's agreeing and PET SNAKE's gluing phase. Subsequently, we describe the operations of the hash table for agreeing a pair of symbols. For the gluing phase, the usage of the hash table differs slightly, and we elaborate on this at the end of this section.

Storing a Value in a DRAM Block DB_ν

A small input logic uses a demultiplexer to select, based on the least significant $\log_2(n_{DRAM}) = 8$ bit of an incoming r-part r, one particular DRAM block DB_ν

where the value r is to be stored. As the number of the DRAM implicitly encodes 8 bits of r already, only $135 - 8 = 127$ bits actually have to be stored in a table entry. Along with each table entry we store a $\log_2(n_{entries}) = 20$ bit *multiplicity* that is to count how often this value has been stored in the hash table. The 5 remaining bits to fill up 19 byte are used as flags, e. g., one flag indicates that the entry is *not empty*, the other bits are used for gluing.

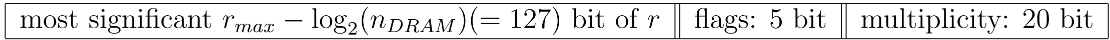


Figure D.1: Hash Table Entry for a Value r

With a size of 19 byte, hash table entries respect byte boundaries, and the address in the DRAM block where to store an incoming r-part r can be obtained by reading off bits no. 8–21 from r. When storing an r-part the multiplicity counter is increased. When the *not empty* flag is set, two cases can occur:

- The r-part in this table entry is identical to the value to be stored: in this case the multiplicity counter is increased.
- The r-part in this table entry differs from the value to be stored: in this case subsequent table entries are searched until a free spot or a table entry with identical r-part is found.

The basic idea of distributing each subtable HT_μ across several DRAM blocks is to ensure that a single DRAM has about $n_{DRAM} = 256$ clock cycles, before a new element to be stored arrives. So simple linear probing should suffice to resolve collisions. To handle irregularities in the distribution of incoming r-parts, we provide safety mechanisms for storing and looking-up elements. These mechanisms are detailed below in a separate subsection.

Looking Up Values in a DRAM Block DB_ν

Once the first subtable HT_0 has been filled with the entries corresponding to a matrix L_i, read queries with r-parts corresponding to a second matrix L_j are received by the hash table. In addition to doing the look-up of these r-parts in HT_0 as described next, all these incoming r-parts are also forwarded to the second subtable HT_1 as a write query: the latter will be filled with these r-parts in exactly the same way as HT_0 has been filled with the r-parts belonging to L_i. This ensures that once the complete list of look-up queries for the second matrix L_j have been processed, the hash table is ready to answer read queries for table HT_1. As the values to be looked-up in HT_1 are exactly the values that have been written in HT_0, when filling HT_0 we also store all incoming r-parts in the order they are arriving in a *buffer DRAM* of size $n_{entries} \cdot r_{max}$ bit.

Remark. The buffer DRAM is realized as a collection of 16 DRAM blocks that are accessed cyclically, therewith accounting for the slower access time of DRAMs.

To check if a queried r-part is stored in a particular DRAM block of a subtable, first the memory address is computed by reading off bits no. 8–21 from the incoming r-part, just as when storing r-parts. Starting at the address obtained in this way, hash table entries are read sequentially until either a read entry is marked as unused—meaning that the queried element is not in the table—or the queried element is found as a table entry. To perform the necessary comparisons for our parameters, we can use a 128-bit comparer which checks in two clock cycles if a stored r-part coincides with the queried r-part. In case a queried value is found in the table, the stored multiplicity is added to a 20 bit *glue size counter*. If the latter counter overflows, this will be reported (see below).

Dealing with a Non-uniform Input Distribution

One could consider a set-up where incoming r-parts are preprocessed to smoothen the bit distribution, e. g., by multiplication with a fixed random matrix. Judging from our software experiments, this does not seem to be necessary, and for simplicity we do without such a preprocessing. Nevertheless, we see from the abovementioned experiments that it can happen that identical r-parts are sent to the hash table in rapid succession. This could be problematic as then a single DRAM block DB_ν might be overwhelmed with a large number of write queries arriving in a (too) short time frame.

Caching repeated write queries. To cope with the problem just mentioned, we place a comparer pipeline of length $n_{duplicates} = 32$ *before* the above-mentioned demultiplexer which directs incoming r-parts to a particular DB_ν. Every pipeline stage can store one *new* r_{max}-bit entry, one *old* r_{max}-bit entry that has already been forwarded to the demultiplexer, an 8-bit multiplicity counter and hosts two r_{max}-bit comparison units, either of which can compare the stored value with an incoming r_{max}-bit entry within two clock cycles. These comparison units are realized as a pipeline of length two and thus can handle a new input every clock cycle. When writing entries into the hash table, and an incoming r-part coincides with the *old* value currently stored in the pipeline stage, this means that the incoming r-part has been sent to the hash table during the last $n_{duplicates}$ queries already. In this case the internal multiplicity counter of the pipeline stage is increased by one, but the r-part is not forwarded to a DRAM block (yet). By setting a simple discard flag, we can ensure that such an r-part is discarded at the end of the pipeline instead of being forwarded to a DB_ν. The *new* entries are forwarded to the next pipeline stage, and the *old* values (including the counters) are shifted into the opposite direction

if one value is handed over to the demultiplexer. If an entry with a stored counter value does not equal zero, the respective entry is handed over to the buffer into the demultiplexer and will be forwarded to the appropriate DB_ν, along with the appropriate counter value.

When looking up entries in the hash table, the above-mentioned discard signal is ignored and all incoming r-queries are forwarded to the individual DRAM blocks DB_ν, which will locally handle repeated r-queries as discussed next.

Local buffering. To cope with repeated read queries, in front of each DRAM block we have another short pipeline of $n_{cache} = 4$ registers, each of which is capable of holding an $r_{max} - \log_2(n_{DRAM}) = 127$-bit value and a 20-bit multiplicity counter: after each read query actually executed by the DRAM, the respective r-part and its multiplicity is stored in one of the pipeline stages. The $n_{cache} = 4$ registers are overwritten cyclically, after each read query executed by the DRAM. If an incoming read query is found in one of the $n_{cache} = 4$ pipeline stages, this means that the answer to the read query is known already and no actual access to the DRAM block is needed. This protects the hash table from being overwhelmed with repeated read queries with identical r-parts. Finally, for the case that different read or write queries reach an individual DRAM block DB_ν in short succession, between the cache registers just mentioned and the actual DRAM block we place $n_{buffer} = 10$ registers, each of which can hold an $r_{max} - \log_2(n_{DRAM}) = 127$-bit entry, a $\log_2(n_{entries}) = 20$ bit counter representing the corresponding column index and an 8-bit counter indicating the number of times this element has been queried in a write-query already while still being held in a pipeline stage. If all n_{buffer} entries in a DRAM run full, a throttle signal is set, indicating that the hash table cannot process further queries at the moment.

Reporting Results

When writing entries into one of the subtables, by default no return value from the hash table needs to be sent, except there is a danger of buffer overrun and the throttle signal is used to prevent the transmission of further write queries.

In the lookup phase no output is produced if a queried r-part is found in the list. The glue size counter is increased only. Once this counter overflows, a warning signal "glue exceeds threshold" is sent. If a queried r-part is not found, then the corresponding column index, which is a $\log_2(n_{entries}) = 20$ bit value, is returned. To keep track of the column indices, a local $\log_2(r_{max})$-bit counter is used that is increased by one whenever a read query is submitted to the hash table.

Gluing Phase

The first step in the gluing phase is identical with the agreeing phase—namely, the r-parts of L_i are stored in the buffer DRAM and sent to HT_0.

In the next step, the r-parts of L_i will be sent to HT_0 again, including the column index, and this time as a look-up query. Of course, each r-part will be found and we store the corresponding 20 bit column index next to the r-part, provided the stored multiplicity for r is not greater than $\sigma_{popular} = 255$. If the r-part is hit for the first time in the second run, the place for the multiplicity counter (in Figure D.1 the value on the right) is used to store the column index, and a flag is set to indicate that this value no longer represents a *multiplicity*. If the *multiplicity* has already been overwritten, the next free space in the hash table is used to store the column index. In case the next entry is not free, the 4 bit flags are used to indicate how many entries to skip. At this next address, there is space for 7 more numbers, and a new forward pointer, if necessary.

In the rare cases, that the multiplicity counter of an r-part is greater than $\sigma_{popular} = 255$, this technique would fill up the DRAM. In this case, the multiplicity is reported through the output control of the chip to the adder when this r-part r is looked up for the first time. Now, at the adder chip, memory for *multiplicity* many counters can be allocated. To simplify the communication, a *popularity number* is assigned to this r by the output control. This number is stored in the DRAM at the place where *multiplicity* has been, and from now on, this *popularity number* is used to identify this particular r. The column index is reported directly to the control unit of the adder when such an r is hit.

Finally, in the third step, the r-parts of L_j, including the column index, will be sent to HT_0 for being looked up. The column index and the list of the column indices of L_i that are stored in HT_0 (or the popularity number) will be reported to the control unit of the adder.

Area Calculation

Part of the incoming data is stored in the buffer DRAM; this can hold up to 2^{20} values with 136 bit. To cope with non-uniform distributions, the inputs are first stored and checked for duplicates in a pipeline with 32 stages. Each stage stores one 156-bit value and one 143-bit values in latches and holds two comparers for 135-bit values (NANDs followed by a tree of XORs). The comparers have a row of 9 latches after stage 4 of the XOR-tree to keep the critical path short. The distribution to the 256 DB_ν requires a tree of 255 demultiplexers (1 to 2 bit) of width 156 and depth 8. Each DB_ν has 10 buffers of width 148 bit to hold the incoming data, realized as latches. Three registers with 148 flip-flops each to store the last and the running requests and three 128-bit compare units and a 20 bit adder. Besides some control logic, there remains the DRAM to store $4 \times 2^{12} \times 152$ bit in each DB_ν.

The buffer DRAM fits in 4 mm^2. The pipeline for checking for duplicates and the demultiplexing can be realized with some 430 000 transistors, 200 000 for the pipeline, 190 000 for the demultiplexing and 40 000 for additional logic. This is an area of approx 0.15 mm^2. Each of the 256 DRAM blocks in a subtable HT_μ requires some 26 000 transistors for buffers (17 500), comparers (6 500) and logic (2000). This corresponds to an area of 0.009 mm^2 for the transistors of each subtable, the DRAM of one block requires 0.063 mm^2. The complete hash table thus fits on an area of 41 mm^2.

Appendix E

Adder

Architecture

The DRAM collection is comprised of 256 DRAMs, each 1 MByte in size. Five 2060-bit buffers are placed at the inputs of each DRAM so as to protect it from multiple fast read/write operations, and five 2048-bit buffers are placed at the outputs so their results can be collected for adding. In front of this arrangement is a small station where a 20-bit value is multiplied to a 20×20 random, fixed, invertible matrix M so as to randomize the input (which in turn randomizes which DRAM will be selected by the input). In all, this comprises the *s-lookup chain*.

Popular r-parts Preprocess

In the glue phase, a matrix L_i is first processed by the hash table and its DRAM buffer is filled with pairs of r-parts and indices. Then it processes its DRAM buffer to assign the numbers of the queries to the r values, and to find *popular r-parts*; these are those r-parts which occur more than 255 times in Pr_{ij}. Since there can be at most 4096 of them, such a popular r-part is assigned a *popularity number* (label it *pid*) from 0 through 4095. When this popularity number is assigned, the multiplicity is known and the appropriate amount of memory is allocated. To keep

track of the memory for the *popular r-parts*, a list of starting point and current write position is needed for each of the up to 4096 popular numbers. The memory for the counters has to be big enough to store up to 2^{20} numbers with 20 bit. This memory is realized as an SRAM with 4 MByte. During the processing of the DRAM buffer, if it encounters a popular r-part, it sends both the popularity number *pid* and the associated index across the MPU bus to the adder. The index is stored at the appropriate memory address, and the current write position for the popularity number is increased. This continues for the duration of the DRAM buffer processing.

DRAM Collection Preprocess

At the same time, the preprocessing of popular r-parts takes place, the adder will also receive the s-part for the current L_i column. The adder will use the column number register and divert that and the s-part to the DRAM collection. The column number will be multiplied to M. The least significant 8 bits of this result are used to select which DRAM to store the s-part in, and the other 12 bits are used to determine its address. Then the s-part is stored. Because of the buffering and randomization, this will in general not slow down the preprocessing, but should an input buffer be filled, a throttle line will signal the traffic controller to stop sending columns in L_i until the writes are processed.

Process

Now columns of L_j are processed. The adder will receive the s-part of the current column of L_j, and it will store it in the adding register. The hash table will send either a collection of indices of where to find the associated s-part of L_i, or a popularity number *pid*. In the first case, the indices are simply marched along the s-lookup chain, multiplied, diverted to the DRAM collection, and the corresponding

s-parts will be output. Each is then added to the value in the adding register, and the result is sent to traffic control for storage. In the second case, the adder takes *pid* and determines the memory address where the numbers start and end. These values are output.

As the s-lookup chain receives indices, it multiplies them to M, obtains results, and uses them in the same way as before to obtain s-parts from the DRAM. These are added to the value in the adding register, and the results are sent to traffic control for storage.

Extraction Stage Process

After equations are generated in the row reducer's `A`-part from the symbol in snake 1, they are sent across the MPU bus one row at a time, and the adder picks them up. The s-lookup chain is bypassed; the adder will store them directly in its SRAM. Then, after the equations from the symbol in snake 2 appear in the row reducer's `A`-part, the adder will send the equations it stored from its SRAM, across the MPU bus, to the row reducer's `A`-part so that those two groups of equations can be row reduced.

If there is a second extraction stage, the row reducer will send the result of its row reduction across the MPU bus, and the adder will store this group of equations (call it E) in its SRAM (deleting the previous group). Then the equations from the symbol in snake 3 are generated (similarly to how those in the symbol in snake 1 were) and stored in the adder's SRAM. Call this group F. Then the equations from the symbol in snake 4 are generated. These will sit in the row reducer's `A`-part. Then the adder sends F back across the MPU bus to the row reducer's `A`-part, and its contents are row reduced. Then, finally, E is sent from the adder's SRAM across the MPU bus to the row reducer's `A`-part, and its contents are row reduced one final

time. Now, a group of equations is at most 2048×2048 bits, yielding $\frac{1}{2}$ MByte, so the two groups E and F will sit comfortably in the 4 MBytes of the SRAM.

Area Calculation

The DRAM collection consumes 0.54 cm^2. The buffers surrounding it contain $256 \times (5 \cdot 2060 + 5 \cdot 2048) = 5\,258\,240$ flip-flops consuming 0.22 cm^2. The SRAM to hold the popular r values consumes some 0.12 cm^2—here we assume that 1T-SRAM can be used and estimate the area equivalent of 1 bit to be 1 transistor. The remaining control logic including the access to the SRAM, updating memory addresses, and the adder (3×2048 flip-flops and 2048 XORs) will not consume more than 0.20 cm^2. Together, this yields an upper bound of 1.1 cm^2 for the adder.

Appendix F

ASIC Implementation Details

To translate the size of a functional unit from its gate count to its transistor count, Table F.1 is used. Note that the D flip-flop number comes from [43, Figure 5.44(a)], and the 2:1 MUX number comes from the four transistors of [43, Figure 5.38] plus two transistors to invert the select signal.

component	AND	OR	NOT	XOR	NAND	NOR	D flip-flop	Latch	2:1 MUX
transistors	6	6	2	6	4	4	12	6	6

Table F.1: Transistor Counts of Logic Gates and Components

To calculate the surface area consumed by each component, we use the numbers in Table F.2, which are obtained by scaling the 90 nm numbers in [42, Table 3.1] down to 45 nm by a factor of 4 resp. the 130 nm numbers there down to 45 nm by a factor 8 (cf. also [26, Table 2]).

	45 nm logic process	45 nm DRAM process
transistor	$0.2975 \mu m^2$	$0.35 \mu m^2$
DRAM bit	$0.0875 \mu m^2$	$0.025 \mu m^2$

Table F.2: Average Surface Area of Hardware Components

If we instead examine more recent data from the *International Technology Roadmap for Semiconductors*, we find that in a 45 nm logic process, the area of a 4-transistor logic gate (with overhead) is 0.71 μm^2 [12, Table 2c], so a single transistor would take on average 0.1775 μm^2. Further, in a 45 nm DRAM Introduction process, 34.36 Gbits can be made on an area of 744 mm^2 [12, Table 2b], so one DRAM bit takes about 0.0216 μm^2. Both of these numbers are less than those in Table F.2, so our estimates are quite conservative. We keep them instead of the more aggressive ITRS numbers to simplify the task of relating results to existing special purpose designs for cryptanalysis.

Bibliography

[1] Gregory V. Bard. *Algorithms for solving linear and polynomial systems of equations over finite fields with applications to cryptanalysis*. PhD thesis, University of Maryland at College Park, Applied Mathematics and Scientific Computation, 2007.

[2] Daniel J. Bernstein. Circuits for Integer Factorization: a Proposal. At the time of writing available electronically at `http://cr.yp.to/papers/nfscircuit.pdf`, 2001.

[3] Andrey Bogdanov, Thomas Eisenbarth, and Andy Rupp. A Hardware-Assisted Realtime Attack on A5/2 Without Precomputations. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems; CHES 2007 Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 394–412. Springer, 2007.

[4] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J.B. Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, Cryptographic Hardware and Embedded Systems – CHES 2007.

[5] Andrey Bogdanov, Marius C. Mertens, Christof Paar, Jan Pelzl, and Andy Rupp. A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2). In *IEEE Symposium on Field-Programmable Custom Computing Machines — FCCM 2006, Napa, CA, USA*, 2006.

[6] Andrey Bogdanov, Marius C. Mertens, Christof Paar, Jan Pelzl, and Andy Rupp. SMITH - A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2). In *2nd Workshop on Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2006*, 2006. Available at `http://www.crypto.ruhr-uni-bochum.de/imperia/md/content/texte/publications/conferences/sharcs2006_matrix.pdf`.

[7] Walter Böhm and Andreas Geyer-Schulz. Exact Uniform Initialization for Genetic Programming. In Richard K. Belew and Michael D. Vose, editors, *Foundations of Genetic Algorithms IV*, pages 379–403. Morgan Kaufman, 1997.

[8] Wieb Bosma, John J. Cannon, and Catherine Playoust. The Magma Algebra System I: The User Language. *Journal of Symbolic Computation*, 24:235–265, 1997.

[9] Johannes Buchmann, Andrei Pyshkin, and Ralf-Philipp Weinmann. A Zero-Dimensional Gröbner Basis for AES-128. In Matthew J.B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop, FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 78–88. Springer, 2006.

[10] Christophe De Canniere, Orr Dunkelman, and Miroslav Knezevič. KATAN and KTANTAN – A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2009*,

volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2009.

[11] Nicolas T. Courtois, Gregory V. Bard, and David Wagner. Algebraic and Slide Attacks on KeeLoq. In Kaisa Nyberg, editor, *Fast Software Encryption, 15th International Workshop, FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.

[12] International Technology Roadmap for Semiconductors. Overall Technology Roadmap Characters. (Key Roadmap Drivers). Available at `http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009Tables_FINAL_ORTC_v14.xls`, 2009.

[13] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. OReilly & Associates, July 1998.

[14] Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, and Colin Stahlke. SHARK: A Realizable Special Hardware Sieving Device for Factoring 1024-Bit Integers. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems; CHES 2005 Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 119–130. Springer, 2005.

[15] Willi Geiselmann, Adi Shamir, Rainer Steinwandt, and Eran Tromer. Scalable Hardware for Sparse Systems of Linear Equations, with Applications to Integer Factorization. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems; CHES 2005 Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2005.

[16] Willi Geiselmann and Rainer Steinwandt. Yet Another Sieving Device. In Tatsuaki Okamoto, editor, *Topics in Cryptology — CT-RSA 2004*, volume 2964 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 2004.

[17] Willi Geiselmann and Rainer Steinwandt. Non-wafer-Scale Sieving Hardware for the NFS: Another Attempt to Cope with 1024-bit. In Moni Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Compute Science*, pages 466–481. Springer, 2007.

[18] Ronald Graham, Donald Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1989.

[19] Tim Güneysu, Timo Kasper, Martin Novotný, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 75(11):1498–1513, November 2008.

[20] David Kahn. *The Codebreakers*. Scribner, 1996.

[21] Auguste Kerckhoffs. La Cryptographie Militaire. *Journal des Sciences Militaires*, 9:5–83 Jan, 161–191 Feb, 1883.

[22] Vladimir Kolmogorov. Blossom V: A new implementatin of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1:43–67, 2009.

[23] Argonne National Laboratory. MPICH2. Available at `http://www.mcs.anl.gov/research/projects/mpich2/`, 2010.

[24] Gregor Leander, Christof Paar, Axel Poschmann, and Kai Schramm. New Lightweight DES Variants. In *Fast Software Encryption*, volume 4593 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2007.

[25] Arjen K. Lenstra and Adi Shamir. Analysis and Optimization of the TWINKLE Factoring Device. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2000.

[26] Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, and Eran Tromer. Analysis of Bernstein's Factorization Circuit. In Yuliang Zheng, editor, *Advances in Cryptology — ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2002.

[27] Sean Murphy and Matt Robshow. Essential Algebraic Structure within the AES. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[28] National Institute of Standards and Technology. *Federal Information Processing Standards Publication 46. Specification for the DATA ENCRYPTION STANDARD (DES)*, January 1977.

[29] National Institute of Standards and Technology. *Federal Information Processing Standards Publication 74. Guidelines for Implementing and Using the NBS Data Encryption Standard*, April 1981.

[30] National Institute of Standards and Technology. *Federal Information Processing Standards Publication 197. Specification for the ADVANCED ENCRYPTION STANDARD (AES)*, November 2001.

[31] Håvard Raddum and Igor Semaev. Solving MRHS linear equations. Cryptology ePrint Archive, Report 2007/285, 2007. Available at `http://eprint.iacr.org/2007/285`.

[32] Håvard Raddum and Igor Semaev. Solving Multiple Right Hand Sides linear equations. *Designs, Codes and Cryptography*, 49:147–160, 2008. Preprint available in [31].

[33] Frank Ruskey. Information on Rooted Trees. The Combinatorial Object Server, University of Victoria, Canada, 2003. Available at `http://www.theory.cs.uvic.ca/~cos/inf/tree/RootedTree.html`.

[34] Ad C.C. Schoonen. Multiple right-hand side equations. Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science, May 2008. Available at `http://alexandria.tue.nl/extra1/afstversl/wsk-i/schoonen2008.pdf`.

[35] Igor Semaev. Sparse Boolean equations and circuit lattices. Presentation at International Workshop on Coding and Cryptography WCC 09, Ullensvang (Norway), May 2009.

[36] Igor Semaev. Sparse Boolean equations and circuit lattices. Cryptology ePrint Archive, Report 2009/252, 2009. Available at `http://eprint.iacr.org/2009/252`.

[37] Adi Shamir. Factoring Large Numbers with the TWINKLE Device. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems. First International Workshop, CHES'99*, volume 1717 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 1999.

[38] Adi Shamir and Eran Tromer. Factoring Large Numbers with the TWIRL Device. In Dan Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2003.

[39] Claude Shannon. Communication Theory of Secrecy Systems. *Bell Systems Technical Journal*, 28-4:656–715, 1949.

[40] Neil J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. AT & T Research Labs, 2009. Available at `http://www.research.att.com/~njas/sequences/`.

[41] Douglas R. Stinson. *Crpytography: Theory and Practice, 2nd Edition*. Chapman and Hall/CRC, 2002.

[42] Eran Tromer. *Hardware-Based Cryptanalysis*. PhD thesis, Weizmann Institute of Science, May 2007. Available at `http://people.csail.mit.edu/tromer/phd-dissertation/`.

[43] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design. A Systems Perspective*. Addison-Wesley Publishing Company, 1st edition, 1985.