THE UNIVERSITY OF CALGARY

# Development of a High-Speed Hybrid Sieve Architecture

by

Kjell Wooding

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF MATHEMATICS AND STATISTICS

CALGARY, ALBERTA

November, 2003

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Development of a High-Speed Hybrid Sieve Architecture" submitted by Kjell Wooding in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Supervisor, Dr. Hugh C. Williams
Department of Mathematics and Statistics

Mark L. Bauer

Dr. Mark Bauer
Department of Mathematics and Statistics

[illegible]

Dr. John Watrous
Department of Computer Science

Jan 20, 2004

Date

# Abstract

A numerical sieve device is an automated device used for solving systems of simultaneous congruences. This thesis describes the design and construction of CASSIE—the Calgary Scalable Sieve—and explains this design in the context of previous sieve devices.

CASSIE employs several key optimizations to the sieve problem, including doubly-focused enumeration, a technique which allow this sieve to achieve sieve rates over $10^6$ times higher than any previous sieve device.

One particular sieve problem—the pseudosquare problem—was examined in detail. Using CASSIE, the table of known pseudosquares was extended to include 12 new values. These values were then used to offer additional computational evidence for a conjecture on the lower bound of the primality proving problem. Additional applications of pseudosquares, including fast, randomized verification of a Rabin-Williams digital signature scheme, and a solution to the unsolicited commercial email (UCE, or Spam) problem are also explored.

# Table of Contents

**Approval Page** ii

**Abstract** iii

**Table of Contents** iv

**Epigraph** x

**1 Introduction to the Generalized Sieve Problem** 1
- 1.1 Motivation and Background . . . 1
- 1.2 The Generalized Sieve Problem . . . 4
  - 1.2.1 Extremes in Sieve Problems . . . 5
  - 1.2.2 Sieve Performance and Measurement . . . 9
  - 1.2.3 Filtering . . . 10
- 1.3 Structure of the Thesis . . . 11

**2 History of Sieve Automation** 12
- 2.1 The Beginnings of Automation . . . 12
  - 2.1.1 The Prototypes . . . 13
  - 2.1.2 E.-O. Carissan's Sieve . . . 14
  - 2.1.3 The Path to Full Automation . . . 15
- 2.2 Lehmer's Sieves . . . 15
  - 2.2.1 The Bicycle Chain Sieve . . . 16
  - 2.2.2 The Photoelectric Sieve . . . 17
  - 2.2.3 The Movie Film Sieve . . . 18
  - 2.2.4 Gérardin's Adding Machine Sieve . . . 19
- 2.3 The Electronic Revolution . . . 20
  - 2.3.1 ENIAC . . . 20
  - 2.3.2 The Proposed Electronic Sieve . . . 22
  - 2.3.3 The Delay Line Sieves: DLS-127 and DLS-157 . . . 23
- 2.4 Software Sieves . . . 25
  - 2.4.1 SWAC . . . 26
  - 2.4.2 The Berkeley IBMs . . . 27
  - 2.4.3 ILLIAC IV . . . 28
- 2.5 Estrin's idea: the Fixed-plus-Variable (F+V) Approach . . . 29
  - 2.5.1 SRS-181 . . . 30
  - 2.5.2 UMSU . . . 31

2.5.3 OASiS . . . . . . . . . . 32
2.5.4 Bronson and Buell (SPLASH) . . . . . . . . . . 33
2.5.5 MSSU . . . . . . . . . . 34
2.5.6 Star Bridge Systems HC 36m . . . . . . . . . . 35
2.6 Software Revisited: Bernstein's Software Sieve . . . . . . . . . . 36

**3 Implementing and Optimizing the Sieve Problem 37**
3.1 Notation and Preliminaries . . . . . . . . . . 37
3.2 Optimizing Sieve Algorithms . . . . . . . . . . 39
3.2.1 The Trivial Sieve Algorithm . . . . . . . . . . 39
3.2.2 Sieve Normalization . . . . . . . . . . 40
3.2.3 Parallelizing the Sieve Problem . . . . . . . . . . 45
3.3 Doubly-Focused Enumeration . . . . . . . . . . 46
3.3.1 The Simultaneous Enumeration Algorithm . . . . . . . . . . 49
3.3.2 Computing the Optimal Bounds . . . . . . . . . . 52
3.4 A Simple Example . . . . . . . . . . 53

**4 The Calgary Scalable Sieve Architecture 56**
4.1 Design Goals . . . . . . . . . . 56
4.2 Approach . . . . . . . . . . 58
4.2.1 Previous Architecture Choices . . . . . . . . . . 58
4.2.2 The Dual-Language Paradigm . . . . . . . . . . 59
4.2.3 The CASSIE Scripting Language . . . . . . . . . . 60
4.2.4 CASSIE Implementation Summary . . . . . . . . . . 62
4.3 Implementation Details . . . . . . . . . . 63
4.3.1 Notes on Algorithms . . . . . . . . . . 63
4.3.2 Sieve Object . . . . . . . . . . 63
4.3.3 Sieve Rings . . . . . . . . . . 64
4.3.4 Ring Creation . . . . . . . . . . 65
4.3.5 Ring Rotation and Normalization . . . . . . . . . . 66
4.3.6 Combination of Ring Operations . . . . . . . . . . 67
4.3.7 Scoreboard . . . . . . . . . . 67
4.3.8 Monitor Object . . . . . . . . . . 68
4.4 Sieve Algorithms . . . . . . . . . . 69
4.4.1 Optimizing the Sieve Algorithm . . . . . . . . . . 69
4.4.2 Optimizing Simultaneous Enumeration . . . . . . . . . . 69
4.4.3 Filters . . . . . . . . . . 72

5 **Sieve Problems and Results** **74**
5.1 The Pseudosquare Problem . . . . . 74
5.1.1 Applications of Pseudosquares . . . . . 75
5.1.2 Pseudosquares and Primality Testing . . . . . 76
5.1.3 Pseudosquare Growth . . . . . 83
5.1.4 Applications to Cryptography . . . . . 85
5.1.5 Applications to Networking — Spam Prevention . . . . . 86
5.2 Pseudosquare Results . . . . . 89
5.2.1 Construction of the Problem . . . . . 90
5.2.2 Numerical Results . . . . . 92
5.3 Minus Class Numbers of Imaginary Cyclic Quartic Fields . . . . . 94
5.3.1 Tabulation Approach . . . . . 98
5.3.2 Construction Approach . . . . . 100
5.3.3 Sieve Results . . . . . 101

6 **Conclusions and Summary** **104**
6.1 Summary . . . . . 104
6.2 Results and Conclusions . . . . . 104
6.3 Future Improvements . . . . . 105
6.3.1 Hardware Improvements . . . . . 105
6.3.2 Software Improvements . . . . . 106

**Bibliography** **107**

A **CASSIE User's Manual** **118**
A.1 Introduction . . . . . 118
A.1.1 Tcl Overview . . . . . 118
A.1.2 Using Objects . . . . . 120
A.1.3 CASSIE Object Hierarchy . . . . . 122
A.2 Sieve Object Detail . . . . . 122
A.2.1 Attributes . . . . . 124
A.2.2 Methods . . . . . 125
A.2.3 Examples . . . . . 129
A.3 Underlying Object Detail . . . . . 130
A.3.1 Ring Object . . . . . 130
A.3.2 Ssieve Object . . . . . 134
A.3.3 Scoreboard Object . . . . . 137
A.3.4 Monitor Object . . . . . 138
A.4 Parallelizing Sieve Problems . . . . . 141
A.5 Sample Sieve Problems . . . . . 142

# List of Tables

3.1 Doubly-Focused Enumeration . . . . . 54

4.1 Negative Pseudosquares . . . . . 62

5.1 Previous Pseudosquare Results . . . . . 93
5.2 New Pseudosquare Results . . . . . 94
5.3 Minimal $p = a^2 + b^2$ with $a = 3$ . . . . . 99
5.4 Minimal $p = a^2 + b^2$ with $a = 9$ . . . . . 100

# List of Figures

4.1 Congruence Conditions for the Negative Pseudosquares, $p \leq 19$ . . . . 59
4.2 CASSIE Script for the Negative Pseudosquares . . . . . . . . . . . . . 61
4.3 Bitfield ring in C . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

5.1 Pseudosquare growth vs. $p$ . . . . . . . . . . . . . . . . . . . . . . . 95
5.2 Pseudosquare growth vs. $n$ . . . . . . . . . . . . . . . . . . . . . . . 96

A.1 UML Class Diagram for Sieve Objects . . . . . . . . . . . . . . . . . . 123

# List of Algorithms

3.1 Simultaneous Enumeration of $x = a_p - a_n$ . . . . . . . . . . . . . . . 50
4.1 Normalize a Ring . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.2 Denormalize a Ring . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.3 Fixed-precision Sieve . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.4 Simultaneous Enumeration of $x = \delta_p - \delta_n$ with Differences . . . . . . 71
4.5 Fixed-Precision Addition with Overflow Check . . . . . . . . . . . . . 72

# Epigraph

It's very esoteric, of course, and since I am practically the only man working in this field you can see how widespread the interest in it is.

—D. H. Lehmer, on sieves [Leh80].

# Chapter 1

# Introduction to the Generalized Sieve Problem

*With the possible exception of the equator, everything begins somewhere.*

—Peter Robert Fleming

## 1.1 Motivation and Background

Though this is not a thesis on factoring integers, the early history of the automated sieve device and the problems of factoring an odd, composite integer, $N$ into non-trivial factors $N = rs$ are closely linked.

The sieve connection comes from an observation of Pierre de Fermat's ([dF94], pp. 256–258); namely that if $N$ is composite and odd, then the factors $r$ and $s$ are also odd, and choosing (arbitrarily) $r < \sqrt{N}$,[1] define:

$$\begin{aligned} U &= \tfrac{s+r}{2} \\ V &= \tfrac{s-r}{2} \end{aligned}$$

Using this definition, $N$ is expressible as a difference of squares, namely $N = U^2 - V^2$ with $\sqrt{N} \leq U < \frac{N+1}{2}$.

With this observation, Fermat reduced the problem of factoring to the problem of finding all values of $x = U^2 - N$ with $U = \left\lfloor \sqrt{N} \right\rfloor + 1, \left\lfloor \sqrt{N} \right\rfloor + 2, \ldots, \frac{N-1}{2}$ for which $x$ is a perfect square. Enumerating successive values of $x$ is quite straightforward by

[1]The trivial case, where $r = \sqrt{N}$ is not considered.

noting that $(x+1)^2 - x^2 = 2x+1$. In other words, by starting with $x = \left\lfloor \sqrt{N} \right\rfloor + 1$, the series of candidates for $x$ may be obtained by adding successive odd integers. This approach to factoring is called the *difference of squares* method and it forms[2] the basis of most modern factoring algorithms.

In addition to this discovery, Fermat noticed the following: it is possible to shorten this search for $x$ by examining the last two digits of $x$ and rejecting any candidates that cannot possibly be a perfect square. As there are only 22 quadratic residues modulo 100, this method succeeds in excluding almost $\frac{4}{5}$ of the choices for $x$.

This basic idea is called *modular exclusion*, and the following observation, due to Gauss, provides a understanding of the power of this technique.

Given $f(x), g(y) \in \mathbb{Z}[x]$, consider the problem of solving a Diophantine equation of the form:

$$f(x) = g(y) \quad \text{for } x, y \in \mathbb{Z} \tag{1.1}$$

Select $k$ exclusion moduli, $M_1, M_2, \ldots, M_k$, that are pairwise relatively prime.[3] For any solution $(x, y)$ of Equation 1.1, the following expression must hold:

$$f(x) \equiv g(y) \pmod{M_i} \quad \text{for } 1 \leq i \leq r$$

Now, for each of the exclusion moduli, $M_i$, determine the acceptable residue classes for $x$ given $y = 0, 1, 2, \ldots, M_i - 1$. In most cases, $x$ will assume a relatively modest set of residue classes (modulo $M_i$). Consider, for example, the Diophantine equation $22 + 97x = y^2$. If this equation is considered modulo 4, the result is

[2] Along with an important modification by Kraitchik

[3] This is not a necessary requirement, but it does serves to simplify the discussion.

$2 + x \equiv y^2 \pmod 4$, and thus $x \pmod 4 \in \{2, 3\}$. Continuing in this fashion with $M_2 = 3, M_3 = 5, M_4 = 7$ produces the following set of congruence criteria:

$$\begin{aligned} x \pmod 4 &\in \{2,3\} \\ x \pmod 3 &\in \{0,2\} \\ x \pmod 5 &\in \{1,2,4\} \\ x \pmod 7 &\in \{0,1,4,6\} \end{aligned}$$

Computing all possible combinations of residues via the Chinese Remainder Theorem (CRT) (discussed in more detail in Section 1.2.1), produces $2 \cdot 2 \cdot 3 \cdot 4 = 48$ possible solutions (modulo 420).[4] Trying each of these candidates in turn quickly reveals a solution for $x = 11$:

$$22 + 97 \cdot 11 = 33^2$$

Problems such as this one, involving systems of simultaneous congruences, are called *sieve problems.* Though this chapter began with a discussion of factoring, it should be clear from this example that the sieve process applies to a more general class of problems. Lehmer [Leh66], once gave the following (by no means exhaustive) list of examples:

1. Find all solutions $(x, y)$ with $x < L$ of the equation $x^2 + D = y^2$ for a given $D \in \mathbb{Z}$.

[4] $x \pmod{420} \in \{6, 11, 14, 27, 39, 42, 62, 71, 74, 99, 102, 111, 119, 126, 134, 146, 147, 162, 167, 174, 179, 182, 186, 207, 231, 239, 242, 246, 251, 266, 267, 279, 287, 291, 294, 302, 314, 326, 342, 347, 351, 354, 371, 386, 399, 407, 414, 419\}$ to be precise.

2. Find the representations of a large number by a given binary quadratic form, $i.e.N = x^2 - y^2$.
3. Find (or count) the integers $x \leq L$ which are power residues for each of a given set of small primes.
4. For a given polynomial, $g$, find (or count) the numbers for which $g(x)$ is divisible by *none* of a given set of small primes.
5. Find the least possible integer value of $g(y)$.
6. Find the binomial units of a given algebraic number field.

## 1.2 The Generalized Sieve Problem

The generalized sieve problem may now be formalized:

**Definition 1.1** *Let* $\mathcal{R} = \{r_1, r_2, ..., r_k\}$ *with* $0 \leq r_j < M$ *for* $j = 1, 2, ..., k$ *be the set of* acceptable *(or* admissible*)* residues modulo $M$. *The tuple* $\{M, \mathcal{R}\}$ *is called a* sieve ring.

**Definition 1.2** *Two sieve rings are said to be* relatively prime *if their modulus values are relatively prime.*

**Definition 1.3** *The* Generalized Sieve Problem (GSP) *is defined in the following manner. Given:*

*1.* $A$, $B \in \mathbb{Z}$ *with* $B > A$ *(the* sieve interval*)*

2. *$k$ sieve rings, $\{M_1, \mathcal{R}_1\}, \{M_2, \mathcal{R}_2\}, \ldots, \{M_k, \mathcal{R}_k\}$, whose moduli, $M_1, M_2, \ldots, M_k$, are relatively prime in pairs.*

*Find all $x \in \mathbb{Z}$ such that $A \leq x < B$ and*

$$\begin{aligned} x \pmod{M_1} \in &\ \mathcal{R}_1 \ \wedge \\ x \pmod{M_2} \in &\ \mathcal{R}_2 \ \wedge \\ x \pmod{M_3} \in &\ \mathcal{R}_3 \ \wedge \\ &\vdots \\ x \pmod{M_k} \in &\ \mathcal{R}_k \end{aligned}$$

*Any solution $x$, satisfying the sieve criteria above is said to be a solution* admitted *by this sieve problem.*

Note that a sieve problem may be thought of as an intersection of sets:

$$\mathcal{S} = \bigcap_{i=1}^{k} \{x \in \mathbb{Z} \mid x \pmod{M_i} \in \mathcal{R}_i \quad A \leq x < B\}$$

**Definition 1.4** *The* width *of a particular* GSP *instance is defined as the number of congruence conditions present in the problem definition;* i.e., *the width of the sieve problem $\mathcal{S} = \bigcap_{i=1}^{k} \{x \in \mathbb{Z} \mid x \pmod{M_i} \in \mathcal{R}_i \quad A \leq x < B\}$ is $k$.*

### 1.2.1 Extremes in Sieve Problems

Generally, a sieve problem is categorized in terms of the density of acceptable residues when compared to the size of the sieve interval. Though this notion of density will be formalized in Section 3.1, the following broad categories of sieve problems will be discussed here:

## Dense Sieve Problems

The *Sieve of Eratosthenes* is perhaps the best known example of a dense sieve algorithm. This method for generating prime numbers dates back over 2300 years. In this algorithm, the primes $\{p_1, p_2, \ldots . p_k\}$ are used as sieve moduli. The acceptable residues are all the residue classes other than zero, *i.e.*$\mathcal{R}_i = \{1, 2, \ldots, p_i - 1\}$. Now, sieving the interval $[p_{k+1}, p_{k+1}^2)$ will reveal all the primes in that range.[5]

For example, given:

$$\begin{array}{rl} x \pmod{2} \in & \{1\} \\ x \pmod{3} \in & \{1,2\} \\ x \pmod{5} \in & \{1,2,3,4\} \\ x \pmod{7} \in & \{1,2,3,5,6\} \\ x \pmod{11} \in & \{1,2,3,4,5,6,7,8,9,10\} \end{array}$$

Sieving over the interval $[13, 169)$ produces the primes 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, and 167.

This type of sieve problem is referred to as a *dense sieve problem*, owing to the large number of candidate solutions in the sieve interval.

## Sparse Sieve Problems

On the other end of the spectrum is a category know as *sparse sieve problems.* In its most extreme case, a problem may be considered with exactly one acceptable residue per sieve modulus. Unlike other forms of the GSP, this case was solved exactly by Sun

[5]This process may be continued indefinitely by adding the newly discovered primes to the list of exclusion moduli and incorporating the appropriate residue conditions.

Tsu over 2000 years ago, and is more commonly known as the Chinese Remainder Theorem (CRT).[6]

**Theorem 1.1** Chinese Remainder Theorem (CRT)

*Given the congruences*

$$\begin{aligned} x \equiv\ & r_1 \pmod{m_1} \\ x \equiv\ & r_2 \pmod{m_2} \\ & \vdots \\ x \equiv\ & r_k \pmod{m_k} \end{aligned}$$

*where* $m_1, m_2, \ldots, m_k$ *are pairwise relatively prime.*

*Set* $M = \prod_{i=1}^{k} m_i$ *and define* $N_i = \frac{M}{m_i}$. *Since* $\gcd(m_i, N_i) = 1$, *the expression*

$$\xi_i N_i \equiv 1 \pmod{m_i} \quad \text{for } i = 1, 2, \ldots, k$$

*is solvable. A solution to these simultaneous congruences is given by*

$$x \equiv \sum_{i=1}^{k} \xi_i N_i r_i \pmod{M}$$

**Proof:** See [HW79], pp. 95. ■

Sparse sieve problems are unusual in that a relatively efficient algorithm (the CRT) exists to determine a solution. This idea motivates a definition that will used extensively in Chapter 3.

[6]In the period when the USA refused to recognize mainland China, D. H. Lehmer would refer to this theorem as the Taiwan Remainder Theorem

**Definition 1.5** *Given a set of sieve rings, $\mathcal{S}_i = \{\mathcal{R}_i, M_i\}$, the* CRT combination *of these rings is defined as the set of solutions obtained via the* CRT *for every possible combination of residues;* i.e.,

$$\mathcal{R} = \left\{ r \in \mathbb{Z} \mid r \equiv \sum_{i=1}^{k} \xi_i N_i r_i \pmod{M}, \quad r_1 \in \mathcal{R}_1, \ldots, r_k \in \mathcal{R}_k \right\}.$$

Notice that when sieve rings are combined in this fashion, the number of acceptable residues in the combined sieve ring may be exactly predicted, as is demonstrated by the following Lemma.

**Lemma 1.1** *The set of acceptable residues, $\mathcal{R}$ obtained via the* CRT *combination of $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k$ with associated residues $M_1, M_2, \ldots, M_k$ (pairwise relatively prime) is exactly $|\mathcal{R}| = \prod_{i=1}^{k} |\mathcal{R}_i|$.*

**Proof:** Consider the sieve ring, $\mathcal{R}$, obtained via the CRT combination of the sieve rings $\mathcal{R}_1$, $\mathcal{R}_2$. As there are only $|\mathcal{R}_1| \cdot |\mathcal{R}_2|$ different ways to combine the acceptable residues from these two sets, it is clear that $|\mathcal{R}| \leq |\mathcal{R}_1| \cdot |\mathcal{R}_2|$.

Suppose $(r_1, \ldots, r_k), (s_1, \ldots, s_k) \in \mathcal{R}_1 \times \cdots \times \mathcal{R}_k$ and

$$\sum_{i=1}^{k} \xi_i N_i r_i \equiv \sum_{i=1}^{k} \xi_i N_i s_i \pmod{M}$$

Then

$$\sum_{i=1}^{k} \xi_i N_i (r_i - s_i) \equiv 0 \pmod{M}$$

so

$$\sum_{i=1}^{k} \xi_i N_i (r_i - s_i) \equiv 0 \pmod{M_i}$$

for each $j = 1, 2, \ldots, k$. As $M_j \mid N_i$ for $i \neq j$, and $\gcd(\xi_i N_j, m_j) = 1$, then the congruence

$$r_j \equiv s_j \pmod{M_j}$$

must hold for all $j = 1, 2, \ldots, k$.

Thus the $|\mathcal{R}| = \prod_{i=1}^{k} |\mathcal{R}_i|$ solutions are all distinct.

■

**Problems of Quadratic Density**

One of the most interesting classes of sieve problems occurs when approximately half of the residues for a given sieve modulus are acceptable. These *problems of quadratic density* are the most frequently encountered sieve problems [Leh53], and occur in investigations such as the examples given in Section 1.1.

### 1.2.2 Sieve Performance and Measurement

In [Pat92], Patterson showed that, when translated into a decision problem, the GSP is NP-complete. In general, there is no known efficient method for solving the GSP. To date, the most successful methods for solving instances of the GSP have involved the construction of devices that automate the search for a solution.

**Definition 1.6** *The* canvass rate *of a given sieve implementation is defined as the number of solutions, $x$, in a given sieve interval $A \leq x < B$ divided by the number of seconds required to sieve the interval. This value is given in terms of* trials per second.

In practice, there is another very important component of a sieve solution that may impact the overall speed of the sieve device.

### 1.2.3 Filtering

It is rare for a sieve problem to be specified in isolation. Quite often, in the reduction from the original problem, a set of additional restrictions are placed on the values $x$ that are admitted by the sieve. Some examples include:

1. $x$ is (is not) a pseudoprime, strong pseudoprime, or provable prime.
2. $x$ is (is not) a perfect square.
3. $x$ is (is not) a perfect cube.
4. A function of $x$ satisfies one of the above conditions (for example, $x^2 + D^2$ is prime).

Though these types of filtering conditions often figure prominently in a particular sieve problem, filtering is not usually considered part of the GSP. Filtering can, however, play a significant role in the efficiency of a particular sieve implementation. For this reason, an additional measure of sieve performance is often defined.

**Definition 1.7** *The* maximum sieving rate *of a particular sieve implementation is defined as the number of values output by the solution detection mechanism for that particular problem.*

**Definition 1.8** *A sieve problem is called* filter-bound *if the maximum sieving rate of the problem is determined by the speed of filtering the outputs.*

*A sieve problem is called* sieve-bound *if the maximum sieving rate of the problem is determined by the canvass rate of the sieve device.*

Note that if a sieve implementation is sieve-bound, the maximum sieving rate is exactly equal to the canvass rate of the device.

## 1.3 Structure of the Thesis

This thesis is concerned with the design and construction of an automated sieve device, the Calgary Scalable Sieve (CASSIE).

Chapter 2 puts the sieve problem in context by outlining the the history of sieve automation. Chapter 3 discusses general techniques for optimizing the sieve problem. Chapter 4 describes the design and implementation of CASSIE. Finally, Chapter 5 describes several applications of the new sieve device, including the pseudosquare problem, its applications, and several record-setting computations obtained using this new device.

# Chapter 2

# History of Sieve Automation

*HISTORY, n. An account mostly false, of events mostly unimportant, which are brought about by rulers mostly knaves, and soldiers mostly fools.*

—Ambrose Bierce

## 2.1 The Beginnings of Automation

Around 1895, F. W. Lawrence rediscovered Fermat's difference of squares method of factoring [Law96]. Recognising its power, he began to consider ways to automate the technique. His first idea was to employ movable paper strips, whose length was a unit multiple of the modulus. By lining up columns of numbers under consideration spaced by the same unit distance, the paper strips (with the appropriate residues marked with a line) could be slowly shifted across these columns and the unacceptable values crossed off. In fact, variants of this columnar idea have been around for as long as mathematicians have been constructing factor tables. Anton Felkel, for example, used a set of 8 rods in 1776 to construct a table of factors up to 408,000.[1] C. F. Hindenburg is said to have used a similar technique even earlier than this [Wil98], though his results were never published.

[1]This table met a rather ignoble end when, after failing to sell, it was collected up and the paper used for cartridges in the war against the Turks.

### 2.1.1 The Prototypes

Lawrence's next idea for sieve automation was much more significant. Recognizing the cyclic nature of the residue conditions, he proposed the construction of a machine with gears representing each of the exclusion moduli, $m_1, m_2, \ldots, m_k$. Each of the $k$ gears would have $m_i$ equally-sized teeth, which would be numbered from 0 to $m_i - 1$. Acceptable residues would be denoted by placing brass studs through the tooth, extending from the gear on both sides. Each of these modulus gears would then be mounted on individual axles in a fan-like arrangement. This arrangement had two purposes. First, it allowed the gears to be driven by a common driving gear. Second, it allowed the mounting hardware to be angled outward so as to allow the protruding brass studs to clear the mounting hardware. Whenever a solution occurred, the protruding studs would make contact with each other, forming a continuous circuit and ringing a bell to notify the operator. By examining the positions of the various modulus rings, or by employing a rotation counter on the driving gear, the operator would be able to determine the value of $x$ at which the solution occurred.

Though Lawrence never built this machine, in 1910 his proposal was translated into French and republished in André Gérardin's journal *Sphinx-Oedipe.* By 1912, at least 3 people had constructed prototypes of Lawrence's machine: Gérardin himself, Maurice Kraitchik, and Pierre Carissan [G1́2]. Though certainly proof of the concept, these early prototypes were not particularly reliable or robust. Encouraged by these early results, however, Pierre Carissan's brother, Eugène-Olivier went on to build a much more precise device; one which deserves to be called the first truly automated sieve device.

### 2.1.2 E.-O. Carissan's Sieve

This automated sieve device was completed by E.-O. Carissan in 1919 [Car20a]. Building on lessons learned in the construction of his brother's prototype, Carissan's sieve was a beautifully machined device consisting of 14 concentric brass rings, each consisting of a driving gear on the bottom, and a set of $m_i$ studs, equally spaced around the circumference of the ring. These 14 rings moduli were chosen to represent the first 17 primes, with $21 = 3 \cdot 7$, $26 = 2 \cdot 13$, $34 = 2 \cdot 17$, $55 = 5 \cdot 11$, 19, 23, 29, 31, 37, 41, 43, 47, and 53 studs respectively.

Each ring was geared to advance at the same linear rate. *i.e.*To advance by one stud for each iteration of a hand-driven counter. A series of 14 contact switches was placed along the radius of the device in a configuration called the *investigation line*. Acceptable residues were indicated by placing a non-conductive cap on the appropriate stud for a particular ring. When a non-conductive cap passed under the investigation line, it raised the switch. When all 14 rings indicated an acceptable residue, a circuit was completed and an audible click could be heard in a telephone headset connected to the circuit and worn by the operator. When this occurred, the device was stopped, and rolled back to the point where the solution occurred, angular momentum having carried the device too far.

The device was driven by hand (though Carissan later indicated plans to add a motor-drive [Car20b]) and featured a 6-digit counter. It was capable of sieving at rates of 35–40 trials per second.

After Carissan's death in 1925, the machine was nearly forgotten. It sat in a drawer at Observatoire de Bordeaux in Floirac, France for nearly 50 years. It was recently rediscovered [SWM95] and currently resides in the Conservatoire Nationale des Arts et Métiers in Paris.

### 2.1.3 The Path to Full Automation

The early sieve prototypes and Carissan's sieve shared two key drawbacks. First, they required a human operator to physically manipulate the machine. This limited the amount of time that could be devoted to a particular sieve problem to either the limit of the operator's attention span, or his arm strength. Second, the solution detection mechanisms required that an operator take note of the solution condition and stop the device. A momentary lapse in the operator's attention could result in a missed solution.

The next wave of automated sieve devices aimed to reduce or eliminate the effects of operator error by improving the devices in two ways:

- Mechanize the advancement of the sieve moduli, to allow the problems to run for extended periods.
- Automate the solution detection mechanism, making it difficult, if not impossible for solutions candidates to be missed or ignored.

## 2.2 Lehmer's Sieves

The next chapter of sieve development can be summed up with one name: Derrick Henry Lehmer. For over 6 decades, Lehmer participated in the design and construction of automated sieve devices.

### 2.2.1 The Bicycle Chain Sieve

Lehmer's first foray into sieve building occurred while he was still an undergraduate at the University of California, Berkeley. In 1927, he adapted an idea of his father's[2] to produce the bicycle chain sieve [Leh28]. Long thought to be the first automated sieve device (Lehmer did not know of Carissan's sieve) the bicycle chain sieve closely resembled Lawrence's original prototype, consisting of 19 loops of bicycle chain suspended from a common drive shaft, and driven by an electric motor. The number of chain links represented particular sieve moduli, combinations of sieve moduli, or powers of sieve moduli: $64 = 2^5, 27 = 3^3, 25 = 5^2, 49 = 7^2, 22 = 2 \cdot 11, 26 = 2 \cdot 13$, and the primes 17, through 67. Acceptable residues were indicated by a pin inserted into the appropriate link, with the $0^{th}$ link painted red to facilitate the counting.

The solution detection mechanism employed by the bicycle chain sieve was very similar to Carissan's design, employing contact switches that were engaged by the pins inserted into acceptable residues. When all switches were lifted (indicating a potential solution had been encountered), a circuit was completed, engaging a relay and stopping the drive motor. A revolution counter on the drive shaft revealed the solution (once the effects of angular momentum were accounted for).

The bicycle chain sieve was capable of sieving at rates of up to 50 trials per second—any faster, and the loose-hanging bicycle chains would tend to bind and become entangled. Despite these relatively low sieve speeds, and the relative difficulty in setting up a sieve problem,[3] the bicycle chain sieve was used on between 50 and 100 computational problems, including the pseudosquare problem (which will

[2]Derrick N. Lehmer, also a professor at U. C. Berkeley
[3]1–2 hours for a typical problem [Wil98]

be revisited in Chapter 5).

Though the whereabouts of the original bicycle chain sieve are unknown (having been disassembled for transport, and subsequently stolen), a replica built by Robert Canepa of Carnegie Mellon is currently in storage at the Computer Museum's History Centre in Mountain View, California [Pom82].

### 2.2.2 The Photoelectric Sieve

Lehmer's next foray into sieve building came in 1932, with the development of the photoelectric sieve. This remarkable machine was modelled closely after Kraitchik's original prototype. It employed 30 modulus gears representing the primes 11–113, and the prime powers $2^6 = 64, 3^3 = 27, 5^2 = 25$, and $7^2 = 49$. Each of these gears was free to rotate independently around one of 2 axles, and was driven by a matched driving gear, chosen to ensure the modulus gear advanced by $1/m_i$ of its circumference with each iteration of the sieve counter. Acceptable residues were indicated by a hole centred below each gear tooth, located a fixed radius from the axle. Unacceptable residues had this hole filled [Leh34].

The photoelectric sieve's solution detection mechanism was the most impressive component of the sieve design, consisting of a light source and an incredibly sensitive photodetector. Light would enter the device at one end, pass through the first 15 modulus gears, and then be reflected via a pair of prisms through the second set of 15 modulus gears and back to the photodetector. The photodetector consisted of a 6-stage amplifier, capable of amplifying the signal received at the photocell by over $7.29 \times 10^8$ times [car33]. If light was detected, indicating a solution candidate, a thermionic relay was tripped, and the electric drive motor stopped.

The photoelectric sieve was capable of operating at speeds of up to 5000 trials per second.

In 1933, the photoelectric sieve was disassembled, and shipped to the Century of Progress Exhibition in Chicago, where Lehmer was hired to demonstrate the device. Unfortunately, the sensitive nature of the photodetector prevented it from being set up, and Lehmer ended up giving his demonstrations on a non-functional sieve. The sieve was never fully reassembled afterwards.

Portions of the photoelectric sieve are still housed at the Computer History Museum in Mountain View, California.

### 2.2.3 The Movie Film Sieve

Despite its then-fantastic speeds, the main failings of the photoelectric sieve were the enormous difficulties involved in setting up a problem, and the problem ensuring the reliable operation of the solution detection mechanism.

Lehmer's next sieve design was intended to address three main concerns: reliability, portability,[4] and ease of use. In 1934, he produced a modification of his bicycle chain idea: the movie film sieve.

This sieve used 18 loops of 16mm film leader draped over a brass drive shaft to represent the sieve moduli. The loop lengths were chosen to be proportional to each of the moduli, and a $\frac{1}{4}$-inch hole was punched in the leader to indicate an unacceptable residue. In a necessary improvement over the original design, adjustable rollers were added to the bottoms of the loops to provide the tension necessary to keep the loops

[4]At the time, Lehmer was in search of employment. The portability criterion ensured that he could continue with his research in the interim.

from slipping. The entire mechanism was driven by an electric motor.

The solution detection mechanism consisted of a series of metallic brushes—one per film loop—that made contact with the brass driving rod through the holes punched in the film. While the device was operating, the drive circuit would remain complete as long as at least one of the brushes remained in contact with the drive shaft. If a solution was encountered, indicated by 18 unbroken loops of film, the circuit would be broken, a relay would trip, and the machine would coast to a stop. As with Lehmer's other designs, the device could then be rolled back to the point of solution, and the solution candidate read from a revolution counter attached to the drive shaft.

Lehmer used this device with some success, even though its top speed was only around 50 trials per second. The key advantage over previous designs was that a particular sieve problem took only about 30 minutes to set up. Unfortunately, the film loops tended to wear out after about 10 hours of use [Leh80] and thus the sieve operation was typically restricted to problems lasting only a few hours in duration.

Like many of Lehmer's other sieve designs, the movie film sieve is currently housed at the Computer History Museum in Mountain View, California.

### 2.2.4 Gérardin's Adding Machine Sieve

In 1937, Gérardin published information on an electric, automated sieve device that he had constructed [G3́7]. As with many of Gérardin's sieve device accounts, few details of its design or construction were supplied. From a photograph of this device, it appears to have been based on an adding machine. Gérardin's account indicated that the device was capable of printing its solutions, and performed around 25,000 operations per working day. Very little else is known about this sieve.

## 2.3 The Electronic Revolution

Mechanical sieve devices have a common failing. Their reliance on moving parts both constrain the speeds of the devices, and make them prone to wear. The advent of electronics and the computer era brought about a revolution in automation that extended to automated sieve devices. In fact, the early history of electronic sieves is closely tied with the early history of computing.

### 2.3.1 ENIAC

The story of sieving on electronic computers began with a family outing on the weekend of July $4^{th}$ in 1946. Where some families might consider a trip to the beach, this particular family outing involved a trip to see the Electronic Numerical Integrator and Calculator (ENIAC) at the Moore School of Electrical Engineering of the University of Pennsylvania.

Taking advantage of some idle time between ballistics calculations,[5] D.H. Lehmer, and his wife Emma set up ENIAC to search for composite numbers, $n$ that divide $2^n-2$ *i.e.*the base-2 Fermat pseudoprimes. The idea was that by verifying and extending the table of known pseudoprimes [Leh36], the task of testing a large number for primality could be reduced to using just Fermat's test with a few small trial divisions.

The algorithm implemented on ENIAC was a fairly straightforward one. For each prime $p$, the computer was to try every value of the exponent $n \leq 2000$ to determine

[5]Making it, quite possibly, the first problem in computational number theory to be solved using "idle time" on available computers.

whether $2^{n-1} \equiv 1 \pmod{p}$. This brute-force method, though seemingly inefficient, could produce a result in less than 2.4 seconds [Leh49].

In the end, Lehmer was able to extend the list of base-2 Fermat pseudoprimes to include all values in the range $10^8$ to $2 \times 10^8$, and produce 85 new factors of $2^k \pm 1$ for $k \leq 500$. This result, in Lehmer's words, was "like picking plums at waist height" [BLS+02].

At the time, ENIAC was still in its original, parallel configuration, involving 20 independent accumulators that could be wired together in a variety of fashions.

Later, on the suggestion of von Neumann, these parallel units were converted to one-word registers, and ENIAC retained just a single accumulator, a model that inspired many computing designs to come.[6]

Though Lehmer's goal was not to implement the GSP on ENIAC, the pseudoprime problem did include a sieve component. For a variety of reasons, including the desire to allow the algorithm to run unattended, the list of primes, $p$, could not be entered as needed via punch cards. Thus, ENIAC had to compute the candidate primes on its own. For the initial list, a sieve process that eliminated all primes $p \leq 47$ was run on some of the accumulators. This sieving success led Lehmer to write a proposal for a fully electronic sieve device [Leh46] capable of sieving at rates far exceeding any of his purely mechanical designs.

[6]A decision which eventually prompted Lehmer to quip
"ENIAC was a highly parallel machine, before von Neumann spoiled it"

### 2.3.2 The Proposed Electronic Sieve

Encouraged by the ENIAC successes, Paul Morton and D.H. Lehmer sat down to develop an electronic sieve, using a series of counters arranged in rings.

Lehmer's proposed electronic sieve used flip-flops implemented with a pair of triode tubes. These flip-flops were arranged as ring counters (each representing a particular sieve modulus), with the output of each flip-flop transferred to the next upon the arrival of a gated clock pulse. Outputs taps were placed from the specific ring moduli specified by the problem instance, and wired to the grid connection of the gate tube (a pentode which effectively acted as an AND gate). The other gate connection for this pentode was wired to the clock signal. The net effect was that if ever a coincidence occurred where none of the output taps feeding the pentode gate showed a signal (*i.e.*, when a solution was encountered), further clock signals were suppressed from reaching the ring counters. Thus, when a solution occurred, the device would stop counting. A regular (decade) counter was employed to count the total number of clock pulses sent, allowing an operator to see if the counting had stopped. Once the solution was recorded, the operator would press a manual switch (which presumably delivered a clock pulse to the ring counters) and the device would continue sieving.

Based on his experiences with ENIAC, Lehmer predicted such a device, if built, could achieve speeds of over 10,000,000 trials per minute [Leh46]. The project ran into difficulty, however, [Leh80] when it became clear that long counters were difficult to construct.[7]

[7] This was likely an early encounter with propagation delays. Bronson and Buell [BB94] described similar issues when designing their Field Programmable Gate Array (FPGA)-based sieve, eventually leading them to exclude the prime 53 from their hardware sieve design.

In the same (unpublished) proposal, Lehmer detailed a possible design for an acoustic sieve capable of sieving 10 times faster than even the electronic sieve. Though neither of these devices was ever fully constructed, the principles of these designs lead Lehmer to conceive of a radically different design, one which eventually became his most successful and reliable sieve device.

### 2.3.3 The Delay Line Sieves: DLS-127 and DLS-157

The Delay Line Sieve (DLS) came online in December, 1965 [BLS$^+$02], and was originally referred to as the DLS-127 [Leh66]. This sieve was fabricated from Navy surplus delay lines—conductors with a stable and known propagation time. A total of 2877 microseconds of delay was available, divided into 31 recirculating "tanks"[8] [Leh80]. Each tank had a pulse shaper and coincidence counter. Pulses were added into the appropriate tank separated by a fixed delay. These pulses would pass by a coincidence counter, and then be reshaped before recirculating again (to overcome the inevitable signal deterioration), effectively providing indefinite operation. The overall length of each pulse tank delay line was proportional to the sieve modulus it was supposed to represent. The set of 31 pulse tanks could therefore be used to represent the primes (or powers of primes) up to 127. The actual moduli used were: 64, 81, 50, 49, 22, 39, 17, 19, 23, 58, 31, and the primes from 37 to 127 [Leh].

Solution candidates were indicated by the simultaneous arrival of pulses at each of the pulse tank coincidence counters. When a solution condition was detected, the device shifted into an *idle mode*. In this mode, each of the pulse tanks were connected

[8]The term "tank" refers to a delay-line loop, and seems to derive its name from Lehmer's unpublished proposal for the acoustic sieve [Leh46], which shared many characteristics with the Delay Line Sieve.

in sequence, with the last tank feeding back into the first. The pulses could cycle in this manner indefinitely. When the device was switched back into the sieve mode, the device would wait until the next multiple of 2877 microseconds, and then the original (circulating tank) behaviour was restored, effectively returning the sieve to the state it was in before solution detection. Problem loading was also done in idle mode, with each new pulse added at the appropriate offset from 2877 microseconds to place it in the desired pulse tank.

The DLS was perhaps Lehmer's most successful sieve design, capable of sieving at speeds of up to $10^6$ trials per second. The success of this design stemmed as much from its reliability,[9] as from its ease of use. Problem setup, for instance, was highly automated. Since most sieving problems can be represented as a function of the form $f(x, y) = 0$, a program was written on Berkeley's IBM 7094 computer that accepted the coefficients to a function of this type, and output a set of punch cards. These cards were then taken to the Bendix G-15 (designed by Harry Huskey) which

[9]The DLS was reliable, but not infallible. John Brillhart recounts the following anecdote: "... I was the first person to run the Delay-Line Sieve, because Lehmer was out of town for 2 weeks at the moment the engineer (Bob Coffin) had finally gotten all the bugs out of the sieve to make it run. Even then, there was one more bug, which showed up when I tried to run the first test problem. You may know that that sieve had an optical reader that read the sieve bit pattern in from a paper tape. The process of making the tape was the following: Dick used the current IBM computer on the campus to do the arithmetic to produce the initial bit string to be loaded into the sieve and had it punched into a bunch of cards. This was carried from the computer center over to the electrical engineering building where Dick's friend Harry Huskey, had one of the computers he had designed, the Bendix G-15, which read the cards and punched the bit pattern onto a paper tape. The tape was then carried to the sieve room, where it was read into the sieve by an optical reader. The tape also had a bit count on it that had to agree with the count of the number of bits the sieve read in.

After I read the tape in and tried the sieve, it didn't work. I phoned Dick, who was in San Diego and told him there was still a problem. He surprised me a great deal by suggesting what to do. He said to connect the two wires on the optical reader in the reverse way, so it would reverse the parity of the bits read. I did it and it worked. I'll never know (I should have asked him) why he thought of that. I suspect it had happened before, and is one of the things that distinguishes people who are all talk with no experience from those who involve real experience with their growing understanding of something." [Bri] would read the cards and output the sieve setup on a long punched tape. Finally, this tape (which could consist of several successive sieve jobs) was taken to the DLS. An optical reader read in the tape values, and automated the process of starting additional jobs (if they were present) when the current job completed.

This job queueing would not be of much value were it not for another innovation: the automatic printing of solutions. Previous sieve designs would stop when a solution was found, waiting to be restarted by the operator. the DLS simply shifted into idle mode, printed the solution, then shifted back into regular sieve operation.

The DLS had both a solution printing mode and a solution counting mode. This was useful in problems with a high solution density, predictable solution densities, and noticeable filtering overhead. To avoid bottlenecks due to filtering, the sieve could be run in solution counting mode for a particular range, and the resulting count of solutions compared against theoretical values. If the two counts differ, the range could be sieved using the solution mode. Otherwise, the process would continue with the next range.

In the early 70s, The DLS was fitted with 6 additional sieve rings, constructed from shift registers, and renamed the DLS-157. The device was retired in 1975 [Wil98] and though it was once believed to be in storage in the Computer History Museum in Mountain View, California, its current wherabouts are unknown.

## 2.4 Software Sieves

Since the early experiments with ENIAC in 1946, there has been sustained interest in the idea of implementing the GSP in software on a general purpose computer. Unfortunately, since von Neumann's early modifications transformed ENIAC into a stored-program computer [Cli48], most general purpose computing designs have adopted a serial approach to the processing of arithmetic operations. This design decision is at odds with the inherent parallelism of the GSP, and hence, purely software-based approaches to solving the GSP have typically lagged behind the dedicated hardware approach. Still, the relatively widespread availability of general purpose computers[10] has assured software solutions an important place in the history of sieve devices. Furthermore, a recent optimization of Bernstein's (see Section 2.6) has rekindled interest in software-based sieve implementations.

### 2.4.1 SWAC

In the Generalized Sieve Problem, a candidate, $x$, is either accepted or rejected by each exclusion modulus, $M$. As this is a binary decision, the GSP seems ideally suited for implementation on a binary computer. This observation was certainly not lost on Lehmer [Bri92] who, while heading the Bureau of Standards' Institute for Numerical Analysis at UCLA, had the opportunity to work on the newly constructed Standards Western Automatic Computer (SWAC), the first large electronic computer to operate in the western United States [BLS+02].

In [Leh53], Lehmer published the first detailed description of a GSP implementation in software. In this implementation, strings of bits were used to represent acceptable residues, 0 indicating acceptable, and 1 indicating not acceptable. These bit strings were then compared 36-bits at a time using a machine operation called EXTRACT. In modern terms, the EXTRACT command worked like a 36-bit logical

[10]Especially when compared to the availability of dedicated sieve devices.

AND operation, with one of its inputs inverted.[11] The use of EXTRACT in this manner allowed 36 bit positions to be examined for solutions in parallel, an idea that later became known as the *multiple solution tap* technique.

If no solution was found, the bit strings were then circularly rotated using a clever two-register multiplication technique, and the process repeated. In this fashion, the SWAC was able to search for solutions at a rate of 1438 per second. In 1954, Lehmer and Selfridge built a 17-ring sieve implementation on the SWAC [Leh54], successfully extending the table of least pseudosquares to include the primes $p \leq 79$.

### 2.4.2 The Berkeley IBMs

In 1967, John Brillhart wrote a software sieve implementation on Berkeley's IBM 7090 to help find factors of integers of the form $2^n \pm 1$ [BS67]. The software, which implemented up to 22 sieve moduli, was capable of sieving at speeds of up to 150,000 trials per second. It accepted a single input, the integer $N$ to be factored, (which had presumably already been tested with Fermat's test to reject any prime, or (rarely) pseudoprime values). It then constructed a target bit string, representing the initial sieving by the first few primes. The remaining moduli were typically combined into double-moduli of the form $m = pq$ with $p$, $q$ representing primes or powers of primes. The bitstrings associated with these moduli would then be repeatedly ANDed with the target. Once all moduli bitstrings were applied, the result was compared with 0. If it matched, meaning no solutions were found, the target bitstring was reintroduced,

[11] EXTRACT was even more versitile than this. The first parameter (the *extractee*) was inverted, and ANDed with the second parameter (the *extractor*). EXTRACT took an optional third parameter (*shift*) which indicated which way, and by how many bit positions to shift the result. For the purposes of the sieve, this third operand was not used and was (presumably) filled with zeros [Hur]. This extra versitility, however, allowed EXTRACT to be used in a variety of logic and floating point applications [Hus97].

and the moduli bitstrings (appropriately rotated) were ANDed again.

If at any point, the target remained nonzero, it meant the existence of a potential solution, which was then tested to determine if a factor had been found [Bri].

This method showed some success, producing several record-setting factorizations for integers of the form $2^n \pm 1$ [BS67].

### 2.4.3 ILLIAC IV

The ILLIAC IV was a unique machine—the first to employ what later came to be known as a Single Instruction Multiple Data (SIMD) architecture. This parallel design allowed each of the 64 Parallel Execution (PE) units (processors) to operate on the same instruction, albeit with different data elements. Though construction began in 1965, the ILLIAC IV did not become operational until 1976.[12]

The architecture featured 64 PE units, connected via 64-bit communications channels arranged in a topology known as a chordal ring [IT89]. This chordal ring allowed PEs to directly communicate with other processors that had logical distances of $\pm1$ or $\pm8$. Each PE had access to a local $2048 \times 64$-bit memory store. A central control unit issued instructions to each of these PEs, and though each of the processors was designed to operate on the same instruction (albeit with data drawn from their local data store), individual processors could be set to selectively "sit out" of a particular operation. This latter flexibility proved so useful it has been part of SIMD architecture design ever since.

[12]Cost overruns and engineering problems plagued the ILLIAC IV project. Though originally planned as a four-node machine, the project was halted after the first node was completed, the costs having ballooned from the original \$8 million estimate to over \$31 million. Several years after its completion, the ILLIAC IV was disassembled, securing its place as one of the largest flops in the history of computing [fol].

In 1976, Lehmer wrote an implementation of the GSP for the ILLIAC IV architecture [Leh76], with each PE representing a particular sieve modulus. In the PE's local RAM, a bitfield representing the acceptable residues for that modulus was created, 0 indicating acceptable, 1 indicating not acceptable. This bitfield was repeated sufficient times to ensure the bit pattern stopped on a word boundary. Once the sieve problem was loaded in this fashion, the current machine word from each of the PEs was logically ORed together. If any bit position of the combined result remained zero, a solution candidate had been found.

The unique parallel architecture of the ILLIAC IV machine allowed Lehmer's GSP implementation to operate with a degree of parallelism not usually possible on general-purpose computers. As a result, Lehmer's sieve implementation was able to reach speeds of 15 million trials per second, making it the second-fastest sieve device Lehmer ever devised. Unfortunately, the experimental nature of the ILLIAC IV machine prevented Lehmer from using it on a long-term basis [SW90].

## 2.5 Estrin's idea: the Fixed-plus-Variable (F+V) Approach

In 1962 [CEFT62], Cantor, Estrin, Fraenkel, and Turn described a new architecture for solving the GSP, based on what they termed a fixed-plus-variable (F+V) design. This revolutionary idea incorporated both a fixed, general-purpose, component, and variable, custom component to produce a device capable of sieving at rates of up to $10^{10}$ numbers per minute. Their proposal outlined three main ideas. First, it described an efficient algorithm for implementing the GSP using shift registers on reconfigurable hardware, including a method for constructing the solution detection mechanism that eliminated the need for the largest modulus ($m_s$).

Second, it described a technique for implementing additional (virtual) sieve rings in software on the attached general purpose computer, by carefully matching the predicted output rate of the shift-register sieve with the arithmetic capabilities of the host machine.

Finally, the paper made the important observation that by employing $r$ solution taps in parallel, and relabelling the bit positions of the 1-bit circular shift registers (effectively converting them to $r$-bit circular shift registers), the parallelism of the sieve device could be increased by a factor of $r$ at the expense of little more than additional solution detection circuitry.

Revolutionary as it was, the Estrin proposal was not acted on for almost 13 years.

### 2.5.1 SRS-181

The first actual implementation of Estrin's fixed-plus-variable (F+V) idea came from Lehmer [Leh80]. Referred to as the SRS-181, and constructed by Lehmer and Morton, this device was similar to the DLS, but used cyclic shift registers in place of delay lines as the variable hardware component of the sieve. The fixed hardware was to be a stand-alone microcontroller device. Unfortunately, before this host device was completed, the sieve device was mistakenly removed from the lab and sold for scrap while Lehmer and Morton were away [Ste89].

Though Lehmer never formally published the device specifications, references in [MB75], and [Leh76], indicate that the Shift Register Sieve was capable of sieving at rates of 20,000,000 trials per second, and like Lehmer's previous designs, the SRS-181 sieve had both search and solution counting modes. The device had 42 hardware rings, implemented using 8-bit shift register TTL integrated circuits, and representing the primes (or prime powers) from $2, 3, \ldots, 181$ [Pat92].

The SRS-181 design was the basis for the first of a series of sieves constructed at the University of Manitoba.

### 2.5.2 UMSU

The next implementation of Estrin's F+V idea was the University of Manitoba Sieve Unit (UMSU), built by Cam Patterson and H. C. Williams in 1983 [Pat83]. This sieve, which began as a reimplementation of the SRS-181, contained 32 rings representing each of the first 32 primes (or prime powers), and was implemented on a set of 3 wire-wrap boards using 500 integrated circuits. It featured 8 solution taps, and a shift rate of 16.67 MHzfor an overall canvas rate of 133,000,000 trials per second.

The sieve acted as a peripheral to a host computer, a PDP-11/45. Software running on the PDP allowed for problem creation, spooling, and filtering, including the implementation of the virtual sieve rings concept first mentioned by Estrin, *et al.*.

The sieve featured automatic checkpointing, where the hardware state was saved and verified for correctness every hour. If a fault was discovered, or the process was interrupted for any reason, it could be restarted without losing more than an hour's work.

Though certainly proof of the usefulness of the shift register design, the next sieve to emerge from the University of Manitoba featured a completely different hardware approach.

### 2.5.3 OASiS

The Open Architecture Sieve System (OASiS), developed by Stephens and Williams in 1985 [Ste89] was the successor to the University of Manitoba Sieve Unit (UMSU). Unlike previous shift-register designs where ring moduli were fixed by the hardware design, OASiS employed a novel RAM-based design that allowed variable-sized moduli to be employed. As this was the first sieve device to allow reconfiguration of the sieve hardware without physical disassembly, OASiS is perhaps one of the best examples of the Estrin's F+V idea.

The variable ring idea was as follows: a sieve pattern, consisting of a bit string of length $m_i$, was replicated $\frac{16}{\gcd(m_i,16)}$ times, and stored contiguously in a 16-bit RAM module. This repetition ensured that the cyclic bit pattern ended exactly on a 16-bit word boundary. Since data stored in the RAM is retrieved one word at a time, this allowed a single RAM access to serve as a 16-tap solution window. To access the next 16 solutions, the RAM index register would be increased by one word (modulo $\text{lcm}(m_i, 16)$).

The original OASiS design featured 16 8192-byte rings. By combining smaller moduli into larger rings (for example, by combining the moduli 5, 7, 11, and 13 into a single ring of size $5 \cdot 7 \cdot 11 \cdot 13 = 5005$), moduli representing the first 37 primes $(2, 3, \ldots, 157)$ could be squeezed into these 16 rings.

OASiS employed a programmable shift rate that, in its fastest configuration, operated at 13.3 MHz. Combined with the 16 solution taps, this allowed for a maximum canvas rate of $2.15 \times 10^8$ trials per second.

In addition to the variable hardware component, OASiS also consisted of a fixed-architecture host system. This host, a MicroVAX II, was responsible for problem setup, filtering, reporting, and verification. Unfortunately, the communication path between OASiS and the host machine was a relatively low-bandwidth 9600 bits-per-second serial port. This meant that problem setup, verification, or checkpointing could take upwards of 10 minutes to complete. A more critical flaw occurred in problems requiring a high degree of solution filtering. Since this filtering was done on the host machine, the low bandwidth communication path became a bottleneck in the sieving process.

Despite these shortcomings OASiS was able to produce a number of impressive sieve results, including the discovery of five previously unknown pseudosquares; $L_{193}, L_{197}, L_{199}, L_{211}$ and $L_{223}$ [SW90].

In 1989, OASiS was fitted with an additional sieve board. The combined device was called OASiS-II, and contained a total of 32 8192-word rings. This improved sieve was able to extend the table of least pseudosquares by an additional 2 primes: $L_{227}$ and $L_{229}$ [LPW95].

### 2.5.4 Bronson and Buell (SPLASH)

The GSP implementation on SPLASH was the first electronic sieve device to employ FPGA technology—reconfigurable hardware devices that essentially allow the construction of application-specific arithmetic units [BB94]. The software-configurability of these FPGA devices make them ideal for realizing the variable component of Estrin's F+V proposal.

The SPLASH hardware consisted of a linear array of 32 Xilinx 3090 FPGA chips, each containing a grid of $16 \times 20$ configurable logic blocks. Each configurable logic block (CLB) could be programmed to assume one of a variety of configurations, including a pair of flip-flops, any single 5-input, 2-output combinatorial function, or two 4-input, 1-output combinatorial functions. For the purposes of the sieve problem, one prime shift register was implemented per FPGA chip. To maximize performance, a 64-bit solution tap was employed, and Estrin's optimization was employed to convert the 1-bit cyclic shift registers to 64-bit cyclic shift register. As originally noted by Lehmer (see Section 2.3.2), long end-around communication paths in shift registers were notoriously unreliable, so wherever possible, short cycles introduced by Estrin's optimization were employed to reduce propagation delays.[13] SPLASH implemented sieve rings for the primes $3, 5, \ldots, 71$ (excluding 53), 97, 127, and 131. Additional sieving (for any combination of primes less than 500), was relegated to a software process running on the host machine.

The SPLASH sieve implementation used 64 solution taps and a master clock rate of 16 MHz, for a total sieve rate of $1024 \times 10^6$ trials per second.

In 1994, Bronson and Buell used this sieve to extend the table of negative pseudosquares, originally published in [LLS70], and later extended in [Ste89].

### 2.5.5 MSSU

The Manitoba Scalable Sieve Unit (MSSU) was another highly successful sieve device. Originally built in 1993 by Lukes, Patterson and Williams [Luk95], this device offered an order of magnitude increase in sieving speed over previous sieve designs, and is still in use today.

[13]Though these short-cycle implementations were constructed for most of the small primes, the modulus 53 did not lend itself well to an efficient shift register implementation. For this reason, sieving for this modulus was left out of the hardware sieve, and relegated to the host machine.

The MSSU hardware consisted of three main components: the sieve controller, the sieve chip array, and the host machine software. The sieve controller accepted commands from the host machine via a standard serial port, and translated the commands into the necessary low-level instructions to operate each of the sieve chip arrays. Sieve chip arrays had 16 sieve chips in each of two slots. The sieve chips themselves were designed using Very Large Scale Integration (VLSI) technology as a 40-pin DIP package, and could each accommodate the first 30 primes. The chips were driven at a shift rate of 24 MHz,[14] and 8 solution taps for an overall sieve rate of $192 \times 10^6$ trials per second. With all 32 sieve chips installed, the MSSU had a maximum theoretical sieve rate of $6.144 \times 10^9$ trials per second.

The final component of the MSSU system was the host machine software. This software was responsible for a wide assortment of tasks, including problem setup, optimization, and spooling. The software also had the ability to apply up to two optional software filters. One of these filters, the virtual ring filter, allowed for the application of congruence conditions that did not fit into the dedicated hardware rings.

### 2.5.6 Star Bridge Systems HC 36m

In 2003, Wake and Buell revisited the FPGA idea with a sieve implementation on their latest generation of F+V hardware: the Star Bridge Hypercomputer 36m [WB03]. This machine employed a dual 2.4 GHz Intel Xeon machine as its fixed host hardware. The variable component consisted of 7 Xilinx Virtex FPGAs, four of which (Xilinx

[14]The chips were originally designed to accommodate a clock rate of 33 MHz, it was later decided to reduce this frequency in order to lower the power consumption, and hence, heat generation of the sieve chips.

XC2V6000's) were available as programmable computing resources.[15]

The overall speed of the HC 36m architecture was limited by the 66 MHz PCI bus used for communication. Synthesis models indicated it was possible to implement 12 sieves per Virtex FPGA, each sieve handling primes up to 151. By employing the most naive form of parallelism (each sieve configured at a different start point), the HC 36m device was capable of 48-bit parallelism. Combined with the 64-bit solution tap, the Star Bridge architecture offered theoretical sieve rate of $192 \times 10^9$ trials per second. For a problem such as the pseudosquare (or negative pseudosquare) problem, a more intelligent optimization could be used, for instance, by combining the residues $8, 3, 5, 7$, and 11 to produce 30 residue classes (modulo 9120). In this configuration, sieve rates of $39 \times 10^{12}$ trials per second could be achieved.

## 2.6 Software Revisited: Bernstein's Software Sieve

Since it had long been shown that sieving with dedicated hardware devices was vastly more efficient than sieving in software on a conventional computer, it came as some surprise when D. J. Bernstein announced in 2000 that he had succeeded in extending the table of pseudosquares (last extended by a 180-day computation by the MSSU) using software running for 10 days on a single-processor general purpose computer, a Pentium IV running at 1406 MHz. His solution used an optimization technique which, though seemingly simple in hindsight, had not previously been applied to sieve designs. This optimization technique, called doubly-focused enumeration, is examined in more detail in Chapter 3.

[15]The other 3 were devoted to handling onboard communications for the HC 36m board.

# Chapter 3

# Implementing and Optimizing the Sieve Problem

*Premature optimization is the root of all evil.*

—Donald E. Knuth

## 3.1 Notation and Preliminaries

Before going any further, some additional properties of sieve problems will now be formalized.

**Definition 3.1** *The sieve problems*

$$\begin{aligned} \mathcal{S}_1 = &\quad \textstyle\bigcap_{i=1}^{r} \{x \in \mathbb{Z} \mid x \pmod{M_i} \in \mathcal{R}_i, \quad A \leq x < B\} \\ \mathcal{S}_2 = &\quad \textstyle\bigcap_{j=r+1}^{s} \{x \in \mathbb{Z} \mid x \pmod{M_j} \in \mathcal{R}_j, \quad A \leq x < B\} \end{aligned}$$

*are* equivalent *if and only if $\mathcal{S}_1 = \mathcal{S}_2$ for all $A, B \in \mathbb{Z}$;* i.e., *for any choice of bounds the set of solutions admitted by each of the sieve problems is the same.*

With this notion of equivalence, the following theorem may now be demonstrated.

**Theorem 3.1** *Given a sieve problem, $\mathcal{S} = \bigcap_{i=1}^{k} \{x \in \mathbb{Z} \mid x \pmod{M_i} \in \mathcal{R}_i, \quad A \leq x < B\}$ of width $k \geq 2$ and whose sieve rings are relatively prime, an equivalent sieve problem of width $k - 1$ can be formed.*

**Proof:** This is a straightforward application of the CRT. Consider the congruences:

$$\begin{aligned} x \pmod{M_1} &\in \mathcal{R}_1 \\ x \pmod{M_2} &\in \mathcal{R}_2 \end{aligned}$$

Let $M = M_1 \cdot M_2$, $N_i = \frac{M}{M_i}$, $\xi_i \equiv N_i^{-1} \pmod{M_i}$ as per Theorem 1.1. Define the set $\mathcal{R}$ to be the CRT combination of all residues from the sets $\mathcal{R}_1$, $\mathcal{R}_2$; *i.e.*,

$$\mathcal{R} = \{r \mid r \equiv \xi_1 N_1 r_1 + \xi_2 N_2 r_2 \pmod{M_1 \cdot M_2}, \quad r_1 \in \mathcal{R}_1, r_2 \in \mathcal{R}_2\}$$

Now, form a new sieve problem, replacing the sieve rings $\{M_1, \mathcal{R}_1\}$ and $\{M_2, \mathcal{R}_2\}$ with the newly constructed ring $\{M, \mathcal{R}\}$. The width of this new sieve problem is $k-1$. By the CRT, $x \in \mathcal{R}$ if and only if $x \in \mathcal{R}_1 \wedge x \in \mathcal{R}_2$. Equivalence of the sieve problems follows from Definition 3.1 ■

**Corollary 3.1** *Any sieve problem* $\mathcal{S} = \bigcap_{i=1}^{k} \{x \in \mathbb{Z} \mid x \pmod{M_i} \in \mathcal{R}_i, \quad A \leq x < B\}$ *consisting of of k relatively prime sieve rings can be replaced by an equivalent sieve problem consisting of a single sieve ring:*

$$\mathcal{S} = \{x \in \mathbb{Z} \mid x \pmod{M} \in \mathcal{R}, \quad A \leq x < B\}$$

*where* $M = \prod_{i=1}^{k} M_i$ *and*

$$\mathcal{R} = \left\{ r \in \mathbb{Z} \mid r \equiv \sum_{i=1}^{k} \xi_i N_i r_i \pmod{M}, \quad r_1 \in \mathcal{R}_1, \ldots, r_k \in \mathcal{R}_k \right\}$$

**Proof:** By Theorem 3.1, any sieve problem of width $k$ may be replaced by an equivalent sieve problem of width $k-1$. This process can be repeated until only a single ring remains. The definitions of $M$ and $\mathcal{R}$ follow from repeated application of the CRT. ■

With this Corollary in hand, the notion of sieve density that was first mentioned in Section 1.2.1 may now be formalized.

**Definition 3.2** *The* density of solutions *for a given sieve problem is defined to be the ratio of acceptable residues to all possible solutions in the sieve interval;* i.e., *for the sieve problem,* $\mathcal{S} = \{x \in \mathbb{Z} \mid x \pmod{M} \in \mathcal{R}, \quad A \leq x < B\}$*, the solution density is given by:*

$$\texttt{density}\,(\mathcal{S}) = \frac{|\mathcal{R}|}{M}$$

Multiplying the sieve density by the size of the sieve interval $(B - A)$ offers a prediction as to the number of solutions that will be obtained by sieving over the indicated range. This prediction is exact when $M \mid (B - A)$.

## 3.2 Optimizing Sieve Algorithms

### 3.2.1 The Trivial Sieve Algorithm

The most obvious algorithm for finding all solutions to an instance of the GSP is to examine each of the values $x = A, A + 1, A + 2, \ldots, B - 1$ sequentially to determine if all of the congruences $x \pmod{m_i} \in \mathcal{R}_i \quad (i = 1, 2, 3, \ldots, k)$ are satisfied.[1]

This approach is also trivially parallelizable, for instance, by partitioning the sieve interval across several sieve units. If $r$ sieve units are available, sieve over the

[1]Bernstein calls this approach *unfocused enumeration* [Ber04].

interval $A \le x < A + \left\lceil \frac{B-A}{r} \right\rceil$ on the first unit, $A + \left\lceil \frac{B-A}{r} \right\rceil \le x < A + 2\left\lceil \frac{B-A}{r} \right\rceil$ on the second, and so forth up to $A + (r-1)\left\lceil \frac{B-A}{r} \right\rceil \le x < B$.

In practice, this trivial algorithm and parallelization tactic are rarely used, as more efficient methods are available. These methods will now be described.

### 3.2.2 Sieve Normalization

In [Leh53], Lehmer described a technique for eliminating single-valued congruences from sieve problems. He called this technique *normalization.*

In general, only one single-valued congruence need be considered when discussing Lehmer's normalization. Consider, for example, the following set of congruence conditions:

$$\begin{aligned} x &\equiv 6 \pmod 8 \\ x &\equiv 2 \pmod 3 \\ x &\equiv 1 \text{ or } 4 \pmod 5 \\ x &\equiv 3, 5 \text{ or } 6 \pmod 7 \end{aligned}$$

The CRT may be applied to the first two congruences in the following manner. Take $M_1 = 8$, $M_2 = 3$, $M = 8 \cdot 3 = 24$, $N_1 = \frac{M}{M_1} = 3$, $\xi_1 \equiv 3^{-1} \equiv 3 \pmod 8$, $N_2 = \frac{M}{M_2} = 8$, and $\xi_2 \equiv 8^{-1} \equiv 2 \pmod 3$, giving:

$$\begin{aligned} x &\equiv \xi_1 N_1 r_1 + \xi_2 N_2 r_2 \pmod M \\ x &\equiv 3 \cdot 3 \cdot 6 + 8 \cdot 2 \cdot 2 \pmod{24} \\ x &\equiv 14 \pmod{24} \end{aligned}$$

More generally, given the residue conditions:

$$\begin{aligned} x \equiv &\ r_1 \pmod{M_1} \quad \wedge \\ x \equiv &\ r_2 \pmod{M_2} \quad \wedge \\ &\vdots \\ x \equiv &\ r_h \pmod{M_h} \end{aligned}$$

An equivalent congruence can be produced:

$$x \equiv r_0 \pmod{m_0}$$

where $m_0 = \prod_{i=1}^{h} M_i$ and $r_0 = \sum_{i=1}^{h} \xi_i N_i r_i \pmod{M_n}$, with $N_i, \xi_i$ defined as per Theorem 1.1.

**Definition 3.3** *The arithmetic progression, $x = ym_0 + r_0$, produced via the* CRT *combination of all single-residue congruences is called the* normalization function *for a particular sieve problem. If $m_0 = 1$, $r_0 = 0$, it is called the* trivial normalization function.

This idea motivates the following definition.

**Definition 3.4** *The* canonical representation *of a sieve problem is defined as the equivalent sieve problem where all single-residue congruences have been combined into a normalization function, $x = ym_0 + r_0$, giving $|\mathcal{R}_i| > 1$ for all remaining sieve rings.*

Lehmer's normalization works as follows. Instead of sieving for solutions, $x$, over the interval $A \leq x < B$, sieve instead for acceptable values of $y$ over the interval:

$$\left\lceil \frac{A - r_0}{m_0} \right\rceil \leq y < \left\lceil \frac{B - r_0}{m_0} \right\rceil$$

Sieve solutions obtained for $y$ may be transformed back into solution for $x$ by applying the normalization function, $x = ym_0 + r_0$, to each of the solutions obtained.

**Theorem 3.2** *Given the canonical sieve problem*

$$\mathcal{S} = \bigcap_{i=1}^{k} \{x \in \mathbb{Z} \mid x \pmod{M_i} \in \mathcal{R}_i, \quad A \leq x < B\}$$

*and a nontrivial normalization vector, $x = ym_0 + r_0$, a sieve problem, $\mathcal{S}^*$, can be found that operates over a smaller sieve interval, but where $\mathcal{S} = \{ym_0 + r_0, y \in \mathcal{S}^*\}$. This sieve problem, called the* normalized sieve problem, *is given by*

$$\mathcal{S}^* = \bigcap_{i=1}^{k} \left\{ y \in \mathbb{Z} \mid y \pmod{M_i} \in \mathcal{Y}_i \quad \left\lceil \frac{A - r_0}{m_0} \right\rceil \leq y < \left\lceil \frac{B - r_0}{m_0} \right\rceil \right\},$$

*where $\mathcal{Y}_i$ is defined as follows:*

$$\mathcal{Y}_i = \{y \mid y \equiv m_0^{-1}(r - r_0) \pmod{M_i}, \quad r \in \mathcal{R}_i\}$$

**Proof:** Consider the sieve problem operating over the interval $A \leq x < B$, with normalization vector $x = m_0 y + r_0$, and satisfying the following residue conditions:

$$\begin{aligned} x \pmod{M_1} &\in \mathcal{R}_1 \\ x \pmod{M_2} &\in \mathcal{R}_2 \\ &\vdots \\ x \pmod{M_k} &\in \mathcal{R}_k \end{aligned}$$

If only the solutions for $x$ lying in the arithmetic progression $x = m_0 y + r_0$ are considered, an alternate set of acceptable residues may be defined as follows. Let

$$\mathcal{Y}_i = \{ y \mid y \equiv m_0^{-1}(r - r_0) \pmod{M_i}, \quad r \in \mathcal{R}_i \}$$

Clearly, $y \in \mathcal{Y}_i$ if and only if $x = ym_0 + r_0 \in \mathcal{R}_i$. Thus, a new sieve problem may be defined as:

$$\begin{aligned} y \pmod{M_1} &\in \mathcal{Y}_1 \\ y \pmod{M_2} &\in \mathcal{Y}_2 \\ &\vdots \\ y \pmod{M_k} &\in \mathcal{Y}_k \end{aligned}$$

Since all acceptable values of $x$ may be obtained from the arithmetic progression $x = ym_0 + r_0$, finding all acceptable $y$ in the interval $\left\lceil \frac{A-r_0}{m_0} \right\rceil \le y < \left\lceil \frac{B-r_0}{m_0} \right\rceil$, will produce all $x$ values in the interval $A \le x < B$.

■

For example, the sieve problem given by $0 \le x < 9240$ and

$$\begin{aligned} x &\equiv 1 \pmod{8} \\ x &\equiv 1 \pmod{3} \\ x &\equiv 1 \text{ or } 4 \pmod{5} \\ x &\equiv 1, 2, \text{ or } 4 \pmod{7} \\ x &\equiv 1, 3, 4, 5, \text{ or } 9 \pmod{11} \end{aligned}$$

can be normalized to produce an equivalent problem operating over a smaller interval.

Evaluating the non-normalized sieve problem over the interval $0 \leq x < 9240$ gives $\mathcal{S} = \{1, 169, 289, 361, 529, 841, 961, 1369, 1681, 1849, 2209, 2641, 2689, 2809, 3481, 3529, 3721, 4321, 4489, 5041, 5329, 5569, 6169, 6241, 6889, 7561, 7681, 7921, 8089, 8761\}$.

By combining the first two congruences into a normalization function $x = 24y+1$, and applying this function to the remaining congruences, new sets of acceptable residues may be obtained:

$$\begin{aligned}
\mathcal{Y}_1 &= \{(r_1 - 1) \cdot 24^{-1} \pmod 5, \quad r_1 \in \{1,4\}\} = \{0,2\} \\
\mathcal{Y}_2 &= \{(r_2 - 1) \cdot 24^{-1} \pmod 7, \quad r_2 \in \{1,2,4\}\} = \{0,1,5\} \\
\mathcal{Y}_3 &= \{(r_3 - 1) \cdot 24^{-1} \pmod{11}, \quad r_3 \in \{1,3,4,5,9\}\} = \{0,1,2,4,7\}
\end{aligned}$$

This leads to the normalized sieve problem:

$$\begin{aligned}
y \pmod 5 &\in \{0,2\} \\
y \pmod 7 &\in \{0,1,5\} \\
y \pmod{11} &\in \{0,1,2,4,7\}
\end{aligned}$$

The normalized sieve interval becomes:

$$\left\lceil \frac{0-1}{24} \right\rceil \leq y < \left\lceil \frac{9240-1}{24} \right\rceil$$

*i.e.* $0 \leq y < 385$. Sieving over this interval produces the solution $\mathcal{S}^* = \{0, 7, 12, 15, 22, 35, 40, 57, 70, 77, 92, 110, 112, 117, 145, 147, 155, 180, 187, 210, 222, 232, 257, 260, 287, 315, 320, 330, 337, 365\}$.

The equivalence of this normalized sieve problem, $\mathcal{S}^*$ and the original sieve problem, $\mathcal{S}$ may be verified by applying the normalization function $x = 24y + 1$ to each $y \in \mathcal{S}^*$.

It should be noted that applying Lehmer's normalization to this sieve problem reduced the effort of computing all solutions in the sieve interval from $B - A$ sieve operations to $\frac{B-A}{m_0}$ sieve operations. This reduction in effort comes at the expense of precomputing the acceptable residues, $\mathcal{Y}_i$, and translating the resulting sieve outputs, $y \in \mathcal{S}^*$ back into values of $x$.

### 3.2.3 Parallelizing the Sieve Problem

There is an obvious optimization method employing Lehmer's normalization technique if multiple sieve units may be used in parallel.

Given a sieve modulus, $M_i$ with $|\mathcal{R}_i|$ acceptable residues, the sieve problem may be partitioned into $|\mathcal{R}_i|$ parallel problems by performing normalization on each of $r_{ij} \in \mathcal{R}_i$ for $0 \leq j < |\mathcal{R}_i|$ acceptable residues, and sieving on each of these problems in parallel; *i.e.*, $x = yM_i + r_{ij}$. The set of sieve results for the original problem then becomes the union of the results for each of the $|\mathcal{R}_i|$ parallelized sieve problems.

This optimization can be useful even if the normalized sieve problems are solved consecutively. As demonstrated by Lehmer [Leh28], an effective speedup of $\frac{M_i}{|\mathcal{R}_i|}$ may still be achieved by executing the normalized sieve problems in series, as each of the $|\mathcal{R}_i|$ normalized sieve problems operates over $\frac{1}{M_i}^{th}$ of the original sieve interval. Lukes [Luk95] calls this optimization *multiple residue optimization.* Bernstein [Ber04] calls it *singly-focused enumeration.*

## 3.3 Doubly-Focused Enumeration

Consider a sieve problem given by: $\mathcal{S} = \bigcap_{i=1}^{k} \{x \in \mathbb{Z} \mid x \pmod{M_i} \in \mathcal{R}_i, \quad A \leq x < B\}$. By repeated application of Theorem 3.1, it is possible to derive an equivalent sieve problem with exactly two sieve rings; *i.e.*, set $M_n = \prod_{i=1}^{s} M_i$, $M_p = \prod_{i=s+1}^{k} M_i$, with $\mathcal{R}_n$, $\mathcal{R}_p$ formed from the CRT combination of $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_s$ and $\mathcal{R}_{s+1}, \mathcal{R}_{s+2}, \ldots, \mathcal{R}_k$ respectively. The new sieve problem is given by:

$$x \pmod{M_n} \in \mathcal{R}_n \quad \wedge \quad x \pmod{M_p} \in \mathcal{R}_p \tag{3.1}$$

Bernstein noted that as a special case of the explicit CRT [BS03], $x$ may be written as the difference of small multiples of $M_n$ and $M_p$; *i.e.*,

$$x = a_p - a_n \;=\; t_p M_n - t_n M_p \tag{3.2}$$

Then, taking this expression modulo both $M_n$ and $M_p$, and combining these congruences with Equation 3.1, the following congruences may be obtained:

$$\begin{aligned} x &\equiv -t_n M_p \pmod{M_n} \in \mathcal{R}_n \\ x &\equiv t_p M_n \pmod{M_p} \in \mathcal{R}_p \end{aligned}$$

By sieving for solutions of $t_p$ and $t_n$ over appropriate intervals, and merging these results according to Equation 3.2, all acceptable values of $x$ in the interval $[A, B)$ may be obtained. Furthermore, solving these two new sieve problems can be vastly more efficient than solving the original sieve problem, $\mathcal{S}$.

This definition may now be formalized as follows:

**Definition 3.5** *Given a sieve problem*

$$\mathcal{S} = \bigcap_{i=1}^{k} \{x \in \mathbb{Z} \mid x \pmod{M_i} \in \mathcal{R}_i, \quad A \leq x < B\}$$

*partition the moduli $M_1, M_2, \ldots, M_k$ into two distinct sets, $\mathcal{M}_n$ and $\mathcal{M}_p$. Define the quantities $M_n$ and $M_p$ as the products of the moduli in these sets; $M_n = \prod_{i=1}^{s} M_i$ and $M_p = \prod_{j=s+1}^{k} M_j$ respectively. A potential solution, $x$, of this sieve problem may be written as the difference of small multiples of these quantities:*

$$x = a_p - a_n = t_p M_n - t_n M_p$$

*This technique is called* doubly-focused enumeration, *and it allows the original sieve problem to be replaced with two equivalent (and usually smaller) ones:*

$$\begin{aligned} \mathcal{S}_p &= \bigcap_{i=1}^{s} \left\{ t_p \in \mathbb{Z} \mid t_p \pmod{M_p} \in \mathcal{T}_p, \quad \left\lceil \frac{A}{M_n} \right\rceil \leq t_p < \left\lceil \frac{B + (M_n - 1)M_p}{M_n} \right\rceil \right\} \\ \mathcal{S}_n &= \bigcap_{i=s+1}^{k} \{ t_n \in \mathbb{Z} \mid t_n \pmod{M_n} \in \mathcal{T}_n, \quad 0 \leq t_n < M_n - 1 \} \end{aligned}$$

*with $\mathcal{T}_p, \mathcal{T}_n$ given by*

$$\begin{aligned} \mathcal{T}_p &= \left\{ r \in \mathbb{Z} \mid r \equiv r_p M_n^{-1} \pmod{M_p}, \quad r_p \in \mathcal{R}_p \right\} \\ \mathcal{T}_n &= \left\{ r \in \mathbb{Z} \mid r \equiv r_n (-M_p)^{-1} \pmod{M_n}, \quad r_n \in \mathcal{R}_n \right\} \end{aligned}$$

*These sets may be combined to produce all acceptable values for $x$ in the range $(A, B]$:*

$$\mathcal{S} = \{ x \mid x = t_p M_n - t_n M_p, \quad t_p \in \mathcal{T}_p, t_n \in \mathcal{T}_n, \quad A \leq x < B \}$$

The equivalence of these sieve problems will now be established.

**Lemma 3.1** *Every $x$ in the range $A \leq x < B$ may be expressed as the difference $x = t_p M_n - t_n M_p$, where $M_p, M_n$ are relatively prime, $\left\lceil \frac{A}{M_n} \right\rceil \leq t_p < \left\lceil \frac{B+(M_n-1)M_p}{M_n} \right\rceil$, and $0 \leq t_n < M_n$.*

**Proof:** Consider the arithmetic progression obtained by fixing $t_n$ and varying $t_p$ in the expression:

$$x = a_p - a_n = t_p M_n - t_n M_p$$

This progression is capable of producing any $x \equiv -t_n M_p \pmod{M_n}$. Thus, if $a_n \pmod{M_n}$ is made to range over all residue classes $\{0, 1, 2, \ldots, M_n - 1\}$, the resulting arithmetic progressions can be used to produce all possible integers $x$ in the interval $[A, B)$ by varying $t_p$.

Consider $t_n \in \{0, 1, 2, \ldots, M_n - 1\}$. Since $\gcd(M_n, M_p) = 1$, it is straightforward to show that $\{t \mid t \equiv t_n M_p \pmod{M_n}, \quad 0 \le t_n < M_n\}$ forms a complete reduced residue system. If not, then for some $0 \le i, j < M_n$, $i \ne j$, the congruence:

$$i \cdot M_p \equiv j \cdot M_p \pmod{M_n}$$

would hold. Multiplying both sides by $M_p^{-1} \pmod{M_n}$, however, gives

$$i \equiv j \pmod{M_n}$$

a contradiction, as $0 \le i, j < M_n$. Hence, it is sufficient to consider $0 \le t_n < M_n$ to produce the necessary arithmetic progressions. Since $t_n$ is always nonnegative, choose $t_p \ge \left\lceil \frac{A}{M_n} \right\rceil$, and as the largest choice for $t_n$ is $M_n - 1$, it follows that $t_p < \left\lceil \frac{B+(M_n-1)M_p}{M_n} \right\rceil$.

■

**Theorem 3.3** *The sets of solutions given by*

$$\mathcal{S} = \bigcap_{i=1}^{k} \{x \in \mathbb{Z} \mid x \pmod{M_i} \in \mathcal{R}_i, \quad A \le x < B\}$$

*and the doubly-focused problems given by Definition 3.5;* i.e.,

$$\mathcal{S} = \{x \mid x = t_pM_n - t_nM_p, \quad t_p \in \mathcal{S}_p, t_n \in \mathcal{S}_n, \quad A \leq x < B\}$$

*are the same.*

**Proof:**

Lemma 3.1 shows that every $x$ in the interval $A \leq x < B$ is expressible as the difference $x = t_pM_n - t_nM_p$, where $M_p = \prod_{i=1}^{s} m_i$ and $M_n = \prod_{i=s+1}^{r} m_i$, $\left\lceil \frac{A}{M_n} \right\rceil \leq t_p < \left\lceil \frac{B+(M_n-1)M_p}{M_n} \right\rceil$, and $0 \leq t_n < M_n$.

Thus if $x \in \mathcal{S}$, write $x = t_pM_n - t_nM_p$, and by Equation 3.1:

$$\begin{aligned} x \equiv & \; -t_nM_p \pmod{M_n} \in \mathcal{R}_n \\ x \equiv & \; t_pM_n \pmod{M_p} \in \mathcal{R}_p \end{aligned}$$

By the construction of $\mathcal{T}_p$ and $\mathcal{T}_n$ in Definition 3.5, it is clear that $t_p \pmod{M_p} \in \mathcal{T}_p \iff t_pM_n \pmod{M_p} \in \mathcal{R}_p$ and $t_n \pmod{M_n} \in \mathcal{T}_n \iff -t_nM_p \pmod{M_n} \in \mathcal{R}_n$ Hence, $x \in \mathcal{S} \iff t_p \in \mathcal{S}_p \wedge t_n \in \mathcal{S}_n$. ∎

### 3.3.1 The Simultaneous Enumeration Algorithm

Though it is clear from Lemma 3.1 that every $x$ in the interval $A \leq x < B$ is expressible as the difference $x = t_pM_n - t_nM_p$, an algorithm to produce these $x$ values from $t_p$ and $t_n$ without retaining all intermediate values is not immediately obvious.

In [Ber01], Bernstein suggests a method of generating these values in a systematic manner, which limits the number of intermediate values that must be retained. He calls this algorithm *simultaneous enumeration* [Ber04], and is given as Algorithm 3.1.

**Algorithm 3.1** Simultaneous Enumeration of $x = a_p - a_n$

```
 1: $first \leftarrow 0$; $last \leftarrow 1$
 2: $\widehat{a_n}[first] \leftarrow \texttt{next}(\mathcal{S}_n) \cdot M_p$
 3: repeat
 4:   $a_p \leftarrow \texttt{next}(\mathcal{S}_p) \cdot M_n$
 5:   $\widehat{x}[first] \leftarrow a_p - \widehat{a_n}[first]$
 6: until $(\widehat{x}[first] \geq A)$
 7: $\widehat{a_n}[last] \leftarrow \texttt{next}(\mathcal{S}_n) \cdot M_p$;    $\widehat{x}[last] \leftarrow (a_p - \widehat{a_n}[last])$
 8: loop
 9:   if $last < |\mathcal{R}_n|$ then
10:     while $(\widehat{x}[last] \geq A)$ do
11:       $last \leftarrow last + 1$
12:       $\widehat{a_n}[last] \leftarrow \texttt{next}(\mathcal{S}_n) \cdot M_p$;    $\widehat{x}[last] \leftarrow (a_p - \widehat{a_n}[last])$
13:     end while
14:   end if
15:   Filter and print $\widehat{x}[first], \ldots, \widehat{x}[last - 1]$
16:   $a_p \leftarrow \texttt{next}(\mathcal{S}_p) \cdot M_n$
17:   if $(a_p > (B + (M_n - 1)M_p))$ then
18:     Quit
19:   end if
20:   for each $i$ from $first$ to $last$ do
21:     $\widehat{x}[i] \leftarrow a_p - \widehat{a_n}[i]$
22:     if $(\widehat{x}[i] \geq B)$ then
23:       first $\leftarrow$ first + 1
24:     end if
25:   end for
26: end loop
```

The algorithm works as follows: the functions $\texttt{next}(\mathcal{S}_p)$ and $\texttt{next}(\mathcal{S}_n)$ return the next output from each of the $\mathcal{S}_p$ and $\mathcal{S}_n$ sieves, respectively. These outputs appear in ascending numerical order.

The loop at line 3 advances the $\mathcal{S}_p$ sieve until the first acceptable $\widehat{x} \geq A$ is obtained.

The main loop starts at line 8, and operates as follows. For a particular acceptable candidate $a_p$, maintain a vector of all permissible candidates $\widehat{a_n}$ (called the *row vector*), such that $\widehat{x} = (a_p - \widehat{a_n}) \geq A$. The loop at line 10 is responsible for appending entries to the end of this row vector. It does so by peeking ahead at the next value for $\widehat{x}$, appending it to the $\widehat{x}$ array if it exceeds the lower bound of the sieve problem ($A$). Since sieve outputs are returned in ascending order, it is sufficient to stop sieving for $\widehat{a_n}$ when $\widehat{x}[last] < A$. Candidate values of $\widehat{x}$ up to, but not including the peek-ahead value are filtered and printed in line 15. The loop at line 20 is responsible for obtaining the next acceptable candidate for $a_p$, and computing the next row vector. In line 23, entries are removed from the front of the row vector if they exceed the upper bound of the sieve interval. Line 17 terminates the algorithm when $a_p$ exceeds the upper bound of the sieve range.

The complexity of this algorithm depends on two factors: the number of solution candidates obtained from the $\mathcal{S}_p$ and $\mathcal{S}_n$ sieves (the number of multiplications), and the average width of the row vector, $\widehat{x} = \{\widehat{x}[\mathit{first}], \ldots, \widehat{x}[\mathit{last} - 1]\}$ (the number of additions per row).

### 3.3.2 Computing the Optimal Bounds

The goal of applying the doubly-focused enumeration technique is to reduce the amount of work required to sieve for $x$ over a particular interval. Thus, the amount of work performed by Algorithm 3.1 must be determined for various problem parameters.

Recall that $M_n = \prod_{i=1}^{s} m_i$ and $M_p = \prod_{j=s+1}^{k} m_j$ are the products of all moduli $m_i \in \mathcal{M}_p$ and $m_j \in \mathcal{M}_n$ respectively. The number of residues admitted by each of the sieves may be determined as follows:

$$\begin{aligned} \texttt{density}\,(\mathcal{T}_p) &= \frac{|\mathcal{R}_n|}{M_n} \\ \texttt{density}\,(\mathcal{T}_n) &= \frac{|\mathcal{R}_p|}{M_p} \end{aligned}$$

To obtain all possible values of $x$, $a_p = t_p M_n$ must be allowed to range over $[A, B + (M_n - 1)M_p)$ and $a_n = t_n M_p$ to range over $[0, (M_n - 1)M_p)$. Thus, the number of solutions for $t_n$ (and hence $a_n$) in the interval $0 \leq t_n < M_n - 1$ is given exactly by $|\mathcal{S}_n| = |\mathcal{R}_n|$. The sieve interval for $t_p$ is given by $\left\lceil \frac{A}{M_n} \right\rceil \leq t_p < \left\lceil \frac{B+(M_n-1)M_p}{M_n} \right\rceil$, and hence there are $\left\lceil \frac{B-A-M_p}{M_n} \right\rceil + M_p$ values in this range. Since this is rarely an exact multiple of $M_p$, a precise prediction of the number of solutions in $\mathcal{S}_p$ may not usually be given. It should be clear, however, that $|\mathcal{S}_p| \approx \left( \left\lceil \frac{(B-A)-M_p}{M_n} \right\rceil + M_p \right) \frac{|\mathcal{R}_p|}{M_p}$. And thus, for a reasonable choices of sieve interval $(B - A)$, the number of solutions is expected to be around $|\mathcal{R}_p|$. Bernstein [Ber04] gives the following approximations for most practical implementations: $(B - A) \approx 10^{20}$, $M_p \approx M_n \approx 10^{14}$.

## 3.4 A Simple Example

Consider the problem of finding all values that satisfy the congruences:

$$\begin{aligned} x \quad &\pmod{24} \equiv 1 \\ x \quad &\pmod{5} \in \{1, 4\} \\ x \quad &\pmod{7} \in \{1, 2, 4\} \end{aligned}$$

By the Chinese Remainder Theorem, any set of solutions to this problem will be cyclic modulo $5 \cdot 7 \cdot 24$. Thus the set of solutions may be completely determined by sieving over the interval $0 \le x < 840$ and testing each choice for $x$ against the three congruences. This approach requires 840 sieve operations.

By normalizing this sieve problem, the effort required to determine the sieve solutions may be reduced by a factor of 24; *i.e.*, $x = 24y + 1$, so sieve for $y$ in the range $\left[\left\lceil\frac{0-1}{24}\right\rceil, \left\lceil\frac{840-1}{24}\right\rceil\right) = [0, 35)$.

For convenience, define the sets $\mathcal{Y}_i = \left\{y \mid y \equiv (r_{ij} - r_0) \cdot m_0^{-1} \pmod{M_i}, \quad r_{ij} \in \mathcal{R}_i\right\}$, where $m_0 = 24$ and $r_0 = 1$. In this case, $\mathcal{Y}_1 = \{0, 2\}$, $\mathcal{Y}_2 = \{0, 1, 5\}$, and the sieve problem becomes:

$$\begin{array}{rcl} y \pmod 5 & \in & \{0,2\} \\ y \pmod 7 & \in & \{0,1,5\} \end{array}$$

for $0 \le y < 35$. Solving this sieve problem would involve 35 sieve operations[2].

Applying both the normalization and doubly-focused enumeration to this problem yields even further improvements. Choose $M_n = 7$, $M_p = 5$. Define $y = t_pM_n - t_nM_p = 7t_p - 5t_n$, and consider this expression modulo both $M_n$ and $M_p$:

$$\begin{array}{rcccl} y & \equiv & 7t_p & \equiv & 2t_p \pmod 5 \\ y & \equiv & -5t_n & \equiv & 2t_n \pmod 7 \end{array}$$

In effect, this can be thought of as two separate normalization problems. In both cases, $m_0 = 2$, $r_0 = 0$. Define:

$$\begin{array}{rlcl} \mathcal{T}_n = & \{r \mid r \equiv r_n \cdot (-5)^{-1} \pmod 7, \quad r_n \in \{0,1,5\}\} & = \{0,4,6\} \\ \mathcal{T}_p = & \{r \mid r \equiv r_p \cdot (7)^{-1} \pmod 5, \quad r_p \in \{0,2\}\} & = \{0,1\} \end{array}$$

Now, simultaneously enumerate over the acceptable residues for both $t_p \in \mathcal{T}_p$ and $t_n \in \mathcal{T}_n$ as per Algorithm 3.1 to form the acceptable choices for $y = t_pM_n - t_nM_p$.

[2]Of course, this figure ignores the work required to normalize and denormalize the sieve problem.

| $t_p \backslash t_n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | **0** | | | | | | |
| 1 | **7** | 2 | | | | | |
| 2 | 14 | 9 | 4 | | | | |
| 3 | 21 | 16 | 11 | 6 | 1 | | |
| 4 | 28 | 23 | 18 | 13 | 8 | 3 | |
| 5 | | 30 | 25 | 20 | **15** | 10 | **5** |
| 6 | | | 32 | 27 | **22** | 17 | **12** |
| 7 | | | | 34 | 29 | 24 | 19 |
| 8 | | | | | | 31 | 26 |
| 9 | | | | | | | 33 |
| 10 | | | | | | | |

*Table 3.1*: Doubly-Focused Enumeration

Table 3.1 offers a more graphical illustration of this enumeration. Acceptable residues for $y$ are highlighted in bold.

Acceptable solutions for $y$ are found by sieving for $t_p$ and $t_n$ over the ranges $0 \leq t_n < 7$ and $\lceil \frac{0}{7} \rceil \leq t_p < \left\lceil \frac{35+(7-1)5}{7} \right\rceil$ respectively. Acceptable solutions occur at the intersection of $t_p \in \{0, 1, 5, 6\}$ and $t_n \in \{0, 4, 6\}$. In other words, $y \in \{0, 5, 7, 12, 15, 22\}$.

The total number of sieving operations required for this optimization is:

$$\begin{aligned} t_n \text{ in } [0, M_n) &= [0, 7) \\ t_p \text{ in } \left[\left\lceil \frac{0}{M_n} \right\rceil, \left\lceil \frac{B+(M_n-1)M_p}{M_n} \right\rceil\right) &= [0, 10) \end{aligned}$$

The total number of sieve operations required above is 17. The doubly-focused technique offers a clear reduction in the sieving effort required to solve this particualr sieve problem. Unfortunately, these savings occur at the expense of 7 modular multiplications, and 8 modular subtractions.[3]

[3]This also ignores the work required to normalize and denormalize the sieve problem

# Chapter 4

# The Calgary Scalable Sieve Architecture

*Things should be made as simple as possible, but not any simpler.*

—Albert Einstein

## 4.1 Design Goals

The Calgary Scalable Sieve (CASSIE) design is based around a simple idea, though one that has plagued engineers and implementors throughout the ages: a system must be usable to be considered successful.

In the context of the sieve problem, the most successful sieve designs have been those with the highest levels of usability. Lehmer's delay line sieve, for example, was generally considered his most successful design, mainly due to its reliability and high degree of automation [Leh66]. Sieve parameters could be entered as coefficients of a problem in the form $f(x,y) = 0$. These coefficients would then be converted automatically[1] to a sieve problem, and printed to a linear tape, ready for input into the DLS.

The photoelectric sieve demonstrated the opposite effect. Though it was Lehmer's fastest sieve design for many years to come, it was rarely used owing to difficulties in setup, problem design, and reliability.[2] The later successes of the Movie Film sieve,

[1]Actually, via an IBM 7090 and the Bendix G-15.

[2]After being disassembled for transport to the Century of Progress Exhibition in Chicago, the photoelectric sieve was never used again.

despite it being almost 50 times slower than the photoelectric sieve, showed that fast problem setup, and reliable behavior were vastly more important measures of success than high speeds.

CASSIE's initial design was driven by three technical design goals:

1. Hybrid Design—CASSIE would employ both FPGA (hardware) and software technology to implement a fixed-plus-variable (F+V) sieve design.
2. High Performance—CASSIE would employ Bernstein's doubly-focused enumeration technique to improve sieve performance.
3. Reliability—CASSIE should be able to run for long periods of time without operator intervention. Detection and recovery from errors should be automatic. Software should be free of memory leaks or crashes.

Experience with early versions of the CASSIE implementation showed that, despite impressive speeds, the sieve design appeared to be following the lead of the photoelectric sieve; that is, it suffered great usability problems. Thus, in July 2003, the underlying CASSIE architecture was changed significantly to reflect a fourth goal:

- Ease of use—Though CASSIE was to incorporate a hybrid software/hardware architecture and advanced optimization techniques, this complexity should be hidden from the user. Sieve problems should be fast, easy, and consistent to set up, regardless of which underlying technologies or optimizations were in place.

## 4.2 Approach

### 4.2.1 Previous Architecture Choices

Previous sieve designs such as the MSSU and OASiS featured a certain amount of automation in the host system implementation. Sieve configuration was flexible, involving a series of simple text-files for job configuration and checkpointing. However, pre- and post-processing of sieve data was by no means automatic, involving additional software and programming libraries to reduce a high-level problem description to its its corresponding congruence conditions. For example, the negative pseudosquare problem [LLS70] can be compactly characterized as follows:

**Definition 4.1** *The* negative pseudosquare problem *is defined as the problem of finding* $N_p \in \mathbb{Z}$ *such that* $N_p \equiv -1 \pmod 8$ *with* $\left(\frac{-N_p}{p_i}\right) = 1$ *for all* $p_i \leq p$.

However, the problem of actually reducing this characterization to the set of congruence conditions (Figure 4.2.1) has to be done by the operator.[3]

In its early implementations, CASSIE was no different from previous sieve designs in this respect. That is, reduction of problems to a GSP instance, and subsequent conversion to a set of doubly-focused congruences was the responsibility of the sieve operator. Unfortunately, the added complexity of producing a set of doubly-focused congruences made sieve problem setup quite tedious, even with the development of automated tools for the purpose.

Thus, in July 2003, the CASSIE design was amended to incorporate a radical design concept: the dual-language approach proposed by Ousterhout [Ous98].

[3]Though once the set of congruences was entered, MSSU and OASiS would automatically normalize any single-residue congruences

$$\begin{array}{lll}
x & \pmod 8 & \in \{7\} \\
x & \pmod 3 & \in \{2\} \\
x & \pmod 5 & \in \{1,4\} \\
x & \pmod 7 & \in \{3,5,6\} \\
x & \pmod{11} & \in \{2,6,7,8,10\} \\
x & \pmod{13} & \in \{1,3,4,9,10,12\} \\
x & \pmod{17} & \in \{1,2,4,8,9,13,15,16\} \\
x & \pmod{19} & \in \{2,3,8,10,12,13,14,15,18\}
\end{array}$$

*Figure 4.1*: Congruence Conditions for the Negative Pseudosquares, $p \leq 19$

### 4.2.2 The Dual-Language Paradigm

The main idea of a dual-language approach for CASSIE is to embed the sieve implementation and control mechanisms into a general purpose scripting language. Manipulation and configuration of sieve parameters is accomplished with the (high-level) scripting language, while the actual sieve implementation is done in whatever low-level language is most suitable.

Whereas with the dedicated tool approach, sieve configuration files contain little more than a textual representation of a congruence problem, in a dual-language approach sieve configuration files can contain high-level programming code. Sieve set-up and configuration then inherits the benefits of a high-level scripting language such as Perl, Tcl, or Python, while retaining the performance advantages of being implemented in a language like C or C++.

For the purposes of CASSIE, John K. Ousterhout's Tool Command Language (Tcl) [Ous94] was chosen as the basis of the script language. Ousterhout is one of the pioneers of the dual-language concept, and Tcl was designed for this purpose. Specifically, it was intended to be integrated with C/C++ in a dual-language capacity.[4]

### 4.2.3 The CASSIE Scripting Language

The CASSIE Scripting Language is a superset of the Tcl scripting laguage that includes both the ability to specify and run sieve problems, and the ability to perform arbitrary-precision arithmeticic operations. Multi-precision arithmetic was implemented using the MPEXPR extension, a high-level interface on top of David Bell and Landon Curt Noll's multi-precision library, *calc* [Nol]. Tcl was further extended with a CASSIE software layer,[5] including a complete software implementation of the GSP, incorporating sieve normalization, doubly-focused enumeration, automatic checkpointing, and an interface to FPGA-based sieves. Integration of the CASSIE library into Tcl was generated via the software wrapper generation tool SWIG [Bea96].

The result is a high-level scripting language well-suited to the specification and manipulation of sieve problems and their results. As an example, the negative pseudosquare problem described in Definition 4.1 was implemented in the CASSIE Script Language as shown in Figure 4.2.3.

As specified, the CASSIE of Figure 4.2.3 searched for all solutions, $x = 24y + 23$ in the range $0 \leq y < 500,000,000$. *i.e.*:

$$0 \leq x \leq 12,000,000,023$$

When this sieve problem was executed the results of Table 4.1 were obtained in

[4]An additional benefit of choosing Tcl is the possibility of (later) adding a graphical user interface via the Tk toolkit.

[5]A complete user's manual for the CASSIE script language is found in Appendix A.

```
#!/usr/local/bin/tclsh

source "sieve-lib.tcl"
# Given a modulus p, return all residues, N, such that the negatives,
# (-N/p) = 1 (i.e. the Negative Pseudosquare problem)
proc lls2_ring {p} {
    set res_l [list]
    for {set i 1} {$i < $p} {incr i} {
        if {[JACOBI -$i $p] == 1} {
            lappend res_l $i
        }
    }
    return [list $p $res_l]
}

sieve lls2
lls2 end 500000000

# Set up reporting / checkpointing
lls2 log lls2.out
lls2 report lls2.rpt

# Combine the 2 single-residue rings into a normalization function
ring r {{8 7} {3 2}}
lls2 normalize [lindex [r get] 0] [lindex [r get] 1]

# Add a ring for each of the primes from 5 ... 19
foreach p [primes 5 19] {
        lls2 ring_add [lls2_ring $p]
}

# Retain the least N_p for each of the primes from 19 to 127
foreach p [primes 19 127] {
    lls2 score_add [lls2_ring $p]
}
# Run the sieve problem
lls2 run
```

*Figure 4.2*: CASSIE Script for the Negative Pseudosquares

| $p$ | $N_p$ |
|---:|---:|
| 19 | 10,559 |
| 23 | 18,191 |
| 29 | 31,391 |
| 31 | 118,271 |
| 37,41 | 366,791 |
| 43,47,53 | 2,155,919 |
| 59,61 | 6,077,111 |
| 67 | 98,538,359 |
| 71 | 120,293,879 |
| 73,79 | 131,486,759 |
| 83 | 508,095,719 |
| 89,97,101,103,107,109 | 2,570,169,839 |

*Table 4.1*: Negative Pseudosquares

approximately 4 seconds on the CASSIE host machine[6].

### 4.2.4 CASSIE Implementation Summary

To summarize, the CASSIE Script Language is extremely powerful tool for setting up and executing sieve problems. It encompasses

- A software sieve implementation in C.
- Object orientation applied by Tcl.
- Automatic (Tcl/C) wrapper code generation using SWIG
- A hardware driver layer in C.
- Arbitrary-precision arithmetic via MPEXPR and *calc*.

[6]Specifically, the code was run on one processor of a dual Athlon 2000+ machine with 2Gb RAM.

- Sieve hardware encapsulation (currently, a Xilinx Virtex 2 2000, Nallatech Ballynuey combination).[7]

## 4.3 Implementation Details

### 4.3.1 Notes on Algorithms

Several algorithms will be presented in this chapter. Variables representing arrays are designated with a hat, as in $\widehat{t_n}$. A specific (numbered) element of an array is designated by an index value contained in square brackets after the array name *i.e.*$\widehat{x}[\mathit{first}]$. Where object orientation is assumed, methods and attributes are indicated by a set of double colons, as in *sieve* :: *start*.

### 4.3.2 Sieve Object

The `sieve` object is the main container for a sieve problem. Most of a user's interaction with CASSIE has to do with the creation and manipulation of `sieve` objects. Based on these attributes, the appropriate parameters of the underlying sieve implementation will be generated automatically.

The `sieve` object encapsulates the following information:

- Sieve type — Whether to use traditional sieve representation, or a doubly-focused one.

- Implementation Type — Whether to use a Software or Hardware implementation.

[7]Hardware details will not be covered in this thesis, as the main idea of the CASSIE script language is to encapsulate and abstract the underlying sieve implementation.

- Parallelism — Whether the sieve instance is standalone, or will be part of a multiprocessor (parallel) implementation.[8]
- Sieve Range — Start, and end values for the sieve problem.
- Normalization — The normalization modulus and residue.
- Problem Filters — Filters to be applied to sieve outputs.

The sieve object also includes links to attached rings, scoreboard, and monitor objects, though this linkage is not normally visible to the user. A detailed user manual describing each of these objects in detail is given in Appendix A.

### 4.3.3 Sieve Rings

Recall from Definition 1.1 that a sieve ring is a tuple consisting of a modulus and its associated acceptable residues, *i.e.*$\mathcal{S} = \{M, \mathcal{R}\}$. In CASSIE, sieve rings are specified as Tcl lists.[9] For example, a ring consisting of a modulus 13, and the acceptable residues 1, 2, 4, and 9, would be denoted:

```
{13 {1 2 4 9}}
```

By Corollary 3.1, any set of sieve rings $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_r$ can be reduced to a single sieve ring $\mathcal{S}$. To this end, CASSIE allows multiple rings to be specified as a list-of-lists. These rings are then combined into a single sieve ring as per this corollary. For example, specifying:[10]

[8]At this time, Message Passing Interface (MPI) is the only supported form of parallelism. This is the mechanism used on the University of Calgary's Advanced Cryptography Laboratory (ACL) cluster.

[9]In Tcl, lists are delimited by curly braces, and entries are separated by whitespace. Lists may be nested to form lists-of-lists, and other compound data structures.

[10]Extra whitespace is ignored in Tcl lists, so this could be equivalently written on one line.

```
struct ring {
        int modulus;
        uint32 *bits; /* bitfield array */
};
```

*Figure 4.3*: Bitfield ring in C

```
{{8 1}
 {3 1}
 {5 {1 4}}
 {7 { 1 2 4}}}
```

as the list of moduli and acceptable residues, to the ring constructor results in the creation of the following, equivalent ring:

```
840 { 1 121 169 289 361 529 }
```

This "magic constructor" is used throughout the CASSIE command language. That is, whenever a sieve ring is expected, a ring list may be specified, and a combined ring will be generated automatically.

### 4.3.4 Ring Creation

When `ring` objects are created in CASSIE, they are converted from the high-level Tcl list format to an internal bitfield structure, indicated in Figure 4.3. The first step in the conversion from a Tcl representation to the bitfield representation is to allocate an array of $m$ machine words. The individual bits of this bitfield are then set to either 1, indicating an acceptable residue, or 0, indicating an unacceptable one. This bitfield representation is then repeated $k$ times, where $k$ is the number of bits in a machine word, in order to make the bitfield end on a machine word boundary.[11] This fully populated bitfield structure is called a `ring` object.

### 4.3.5 Ring Rotation and Normalization

The process of associating a `ring` object with a sieve problem is called *attaching* a ring to the sieve problem. Before a ring may be attached to a `sieve` object, however, it may need to be rotated, and/or normalized, depending on the sieve parameters.

In the context of a sieve problem, the *position* of a ring indicates the residue (or in the case of multiple solution taps, residues) currently under investigation. Changing the start value for a sieve problem is akin to *rotating* each of the attached rings to that new start value modulo each of the ring moduli.

Normalization, as described in Section 3.2.2, is a process by which the acceptable residues are permuted and shifted in order to achieve a speedup in sieve operation. This transformation may be accomplished using the following algorithm, noted in [Luk95].

A similar algorithm (Algorithm 4.2) may be developed to accomplish denormalization. In this manner, the appropriate permutation may be generated by first locating the bit position of the final (*ring* :: *modulus* − 1) entry in the sieve, and repeatedly subtracting the normalization modulus. Thus:

$$y_k \equiv Mx_k + R \equiv M(M_r - 1) + R \equiv R - M \pmod{M_r}$$

[11]Since the moduli of interest are usually odd and machine word sizes are usually powers of two, $km$ is a crude approximation to the more correct value, $lcm(k, m)$.

**Algorithm 4.1** Normalize a Ring

```
Variables:
  $r, i$ : $UInt32$
  $newring$ : $Ring$
Inputs:
  $modulus, residue$ : $UInt32$
  $ring$ : $Ring$
Outputs:
  $newring$ : $Ring$
Algorithm:
1: $newring :: modulus \leftarrow ring :: modulus$
2: $r \leftarrow residue \bmod ring :: modulus$
3: for $i \leftarrow 0$ to $(ring.modulus - 1)$ do
4:     $newring :: residues[i] \leftarrow ring :: residues[r]$
5:     $r \leftarrow (r + modulus) \pmod{ring :: modulus}$
6: end for
7: return $newring$
```

### 4.3.6 Combination of Ring Operations

When the normalization and rotation transformations are combined, it is important to decide in which order these operations will occur.[12] In CASSIE, normalization occurs before rotation. In other words, the *start* and *end* attributes of a `sieve` object refer to the start and end values of the normalized sieve problem. The effective sieve interval for the original (non-normalized) problem may be computed via:

$$\begin{aligned} start_{eff} &= m_0 \cdot start + r_0 \\ end_{eff} &= m_0 \cdot start + r_0 \end{aligned}$$

where $m_0$ and $r_0$ are the normalization modulus and residue respectively.

[12]In general, the rotation and normalization transformations do not commute.

**Algorithm 4.2** Denormalize a Ring

```
Variables:
  $lastm, i : UInt32$
  $newring : Ring$
Inputs:
  $modulus, residue : UInt32$
  $ring : Ring$
Outputs:
  $newring : Ring$
Algorithm:
1: $newring :: modulus \leftarrow ring :: modulus$
2: $lastm \leftarrow residue - modulus \pmod{ring :: modulus}$
3: for $i \leftarrow (ring :: modulus - 1)$ downto 0 do
4:     $newring :: residues[lastm] \leftarrow ring :: residue[i]$
5:     $lastm \leftarrow (lastm - modulus) \pmod{ring :: modulus}$
6: end for
```

### 4.3.7 Scoreboard

The `scoreboard` object implements a variant of the the virtual ring filter (common to many previous F+V sieve designs) where values emerging from the sieve are *partially matched* against additional sieve rings. To implement partial matching, scoreboard sieve rings are ordered by modulus, and candidates emerging from the sieve are tested against each of the scoreboard moduli in turn. Each scoreboard ring remembers the smallest candidate that has met its acceptable residue criteria. If the new candidate is smaller than this value and meets the residue criteria, the "best" value is updated. If the candidate fails either of these conditions for a particular scoreboard modulus, no further matching is performed.

In this fashion, the scoreboard object keeps track of the smallest solution candidates for each of a set of sieve moduli that are not included in the original sieve problem (for example, the results summarized in Table 4.1). This behavior is especially useful in doubly-focused sieves, as solution candidates emerge in essentially random order.

### 4.3.8 Monitor Object

The `monitor` object encapsulates all log and checkpointing information associated with a sieve problem. In particular, it specifies the log and checkpoint filenames, the frequency of checkpointing, and whether the sieve problem should terminate after a set number of checkpoints are reached.

## 4.4 Sieve Algorithms

### 4.4.1 Optimizing the Sieve Algorithm

A basic sieve algorithm is shown in Algorithm 4.3. Clearly, there is not much in the basic algorithm that can be optimized. The speed of this algorithm is limited by the speed of the sieve operation in line 3, and the speed of the filtering algorithm in line 10 in place. Filtering is discussed in Section 4.4.3.

**Algorithm 4.3** Fixed-precision Sieve

```
1:  $x \leftarrow start$
2:  loop
3:      $\delta \leftarrow \texttt{nextd}(\mathcal{S})$
4:      $nx \leftarrow\!\!\!\!+\; x + \delta$
5:      $x \leftarrow nx$
6:      if $(nx < \delta)$ or $(nx \geq end)$ then
7:          Quit
8:      end if
9:      Checkpoint (if necessary)
10:     Filter and Test $x$
11: end loop
```

### 4.4.2 Optimizing Simultaneous Enumeration

The simultaneous enumeration algorithm introduced in Chapter 3 (Algorithm 3.1) is a means of iterating over the sieve solutions $t_n, t_p$, to produce all the values of $x = t_pM_n - t_nM_p$ in such a way that a relatively modest number of intermediate values need be retained. In this section, some modest improvements to the simultaneous enumeration algorithm will be presented.

A key optimization is as follows. In the original description of the simultaneous enumeration algorithm, the functions $\texttt{next}(\mathcal{S}_p)$ and $\texttt{next}(\mathcal{S}_n)$ returned the next output from each of the $\mathcal{S}_p$ and $\mathcal{S}_n$ sieves, respectively. By modifying the sieve routines to return *differences* between successive sieve outputs, it may be possible to avoid many of the multiplications of the original algorithm. *i.e.*Rather than returning the sieve sequence $x \in \{x_1, x_2, x_3, \ldots\}$, return the sequence $x \in \{x_1, (x_2 - x_1), (x_3 - x_2), \ldots\}$. Since sieve output values obtained via differences will be much smaller than the original sieve outputs, precomputation or caching of small multiples of $M_n$ and $M_p$ can vastly improve sieve performance. This modification is shown in Algorithm 4.4.

#### Algorithm Notes

In line 1, *start* indicates the start value of the sieve $\mathcal{S}_p$. This ensures that the first value assigned to $\delta_p$ (line 4) is absolute, and thus $t_p = \sum \delta_p$ is true at all times.

The terminating condition at line 17 is actually quite difficult to implement as written. Recall from Section 3.3 that $t_pM_n$ ranges from $A$ to $B+(M_n-1)M_p$, so there are $T_p = \left\lfloor \frac{B-A+(M_n-1)M_p}{M_n} \right\rfloor$ different choices for $t_p$ in this range. From the discussion following Definition 3.2, the number of solutions admitted by the sieve $\mathcal{S}_p$ may be exactly predicted only when number of choices for $t_p$ divides the sieve modulus,

**Algorithm 4.4** Simultaneous Enumeration of $x = \delta_p - \delta_n$ with Differences

```
1: $\mathit{first} \leftarrow \mathit{last} \leftarrow 1$; $\mathit{rows} \leftarrow 0$; $\delta_p \leftarrow \mathit{start}$
2: $\widehat{\delta_n}[\mathit{first}] \leftarrow \texttt{nextd}(\mathcal{S}_n) \cdot M_p$
3: repeat
4:    $\delta_p \leftarrow \delta_p + \texttt{nextd}(\mathcal{S}_p) \cdot M_n$;    $\mathit{rows} \leftarrow \mathit{rows} + 1$
5: until $(\delta_p \geq A + \widehat{\delta_n}[\mathit{first}])$
6: $\widehat{x}[\mathit{first}] \leftarrow \delta_p - \widehat{\delta_n}[\mathit{first}]$
7: $\mathit{next} \leftarrow \mathit{last} + 1$
8: $\widehat{\delta_n}[\mathit{next}] \leftarrow \texttt{nextd}(\mathcal{S}_n) \cdot M_p$
9: loop
10:    while $(\mathit{next} < |\mathcal{S}_n|)$ and $\left(\widehat{x}[\mathit{last}] \geq A + \widehat{\delta_n}[\mathit{next}]\right)$ do
11:       $\widehat{x}[\mathit{next}] \leftarrow \widehat{x}[\mathit{last}] - \widehat{\delta_n}[\mathit{next}]$
12:       $\mathit{last} \leftarrow \mathit{next}$; $\mathit{next} \leftarrow \mathit{next} + 1$
13:       $\widehat{\delta_n}[\mathit{next}] \leftarrow \texttt{nextd}(\mathcal{S}_n) \cdot M_p$
14:    end while
15:    Filter and print $\widehat{x}[\mathit{first}], \ldots, \widehat{x}[\mathit{last}]$
16:    $\delta_p \leftarrow \texttt{nextd}(\mathcal{S}_p) \cdot M_n$;    $\mathit{rows} \leftarrow \mathit{rows} + 1$
17:    if $(\mathit{rows} \geq |\mathcal{S}_p|)$ then
18:       Quit
19:    end if
20:    for each $i$ from $\mathit{first}$ to $\mathit{last}$ do
21:       $\widehat{x}[i] \leftarrow \widehat{x}[i] + \delta_p$
22:       if $(\widehat{x}[i] \geq B)$ then
23:          first $\leftarrow$ first $+ 1$
24:       end if
25:    end for
26: end loop
```

$i.e. T_p | M_p$. Since this is rarely the case[13], a more practical stopping condition is required. One possibility is to test whether $t_p = \sum \delta_p < \left\lceil \frac{B-A+(M_n-1)M_p}{M_n} \right\rceil$.

If this algorithm is implemented using fixed-precision integers,[14] and $B$ is chosen to be the largest representable integer, care must be taken to ensure the comparison at line 22 checks for overflow in the addition on line 21. The usual approach is to use a construct such as is shown in Algorithm 4.5.

**Algorithm 4.5** Fixed-Precision Addition with Overflow Check

```
1: $x \leftarrow \widehat{x}[i] + \delta_p$
2: $\widehat{x}[i] \leftarrow x$
3: if $(\widehat{x}[i] \geq B)$ or $(x < \delta_p)$ then
4:     first $\leftarrow$ first $+ 1$
5: end if
```

### 4.4.3 Filters

Filters are C functions that may be used to eliminate solution candidates of a certain form once they have emerged from the sieve. CASSIE includes several standard filters:

- perfect_square — Reject $x$ if it is a perfect square.
- perfect_cube — Reject $x$ if it is a perfect cube.
- abprime — Reject $x$ unless $x^2 + a^2$ is a prime, for some parameter $a$.
- probprime — Reject $x$ if it fails a probable primality test.
- none — Perform no filtering of results.

[13]The size of the sieve interval, $H = B - A$, is usually determined by factors such as the largest integer expressible by the underlying hardware; typically $2^{32} - 1$ or $2^{64} - 1$.

[14]Fixed-precision integers are taken to mean integers implemented using a fixed word size, $k$. Operations on these integers are assumed to be valid modulo $2^k$.

Many sieve problems define filtering requirements that are more specific than have been provided here. For this purpose, an Application Programming Interface (API) has been provided for the purpose of developing new filters, and is discussed in Appendix A.

# Chapter 5

# Sieve Problems and Results

*However beautiful the strategy, you should occasionally look at the results.*

—Sir Winston Churchill

## 5.1 The Pseudosquare Problem

The pseudosquare problem, first considered by Kraitchik [Kra24] is characterized in the following manner:

**Definition 5.1** *Given an odd prime, $p$, a* pseudosquare, *$L_p$, is defined as the least positive integer satisfying:*[1]

- $L_p \equiv 1 \pmod 8$
- *The Legendre symbol* $\left(\frac{L_p}{q_i}\right) = 1$ *for all odd primes* $q_i \leq p$
- *$L_p$ is not a perfect square.*

In other words, the pseudosquare, $L_p$ behaves (locally) like a perfect square modulo all small primes $q \leq p$,

Kraitchik originally provided pseudosquare results up to $L_{47}$ in [Kra24], pp. 41–46. Since that time, various efforts, spearheaded first by D. H. Lehmer ([Leh54])

[1]In [LLS70], and [BB94], pseudosquares are defined as *any* nonsquare integer satisfying the conditions of Definition 5.1. Pseudosquares satisfying the least positive integer criterion are referred to in these works as *least pseudosquares*.

and later by H. C. Williams ([LPW95]) extended this list to $L_{277}$. To this point, optimization techniques for the pseudosquare problem remained largely unchanged, and advances involving sieve results were made mainly through the construction of improved (faster) sieve devices. In 2000, however, Bernstein's publication ([Ber04]) of the doubly-focused sieve technique (and $L_{281}$) breathed new life into the problem.

It should be noted that the term *pseudo-square* has a different and unrelated definition given by Atkin in [Atk65] (and again in [BR98]). Atkin's pseudo-squares will not be considered in this thesis.

### 5.1.1 Applications of Pseudosquares

The growth rate of pseudosquares has several key applications in Number Theory. Hall, in [Hal33] proved the following result:

**Theorem 5.1** *A number which is a quadratic residue of every prime not dividing it is a perfect square.*

A natural consequence of this theorem is that the values for $L_p$ tend to infinity with $p$. A variant of this result, where least *prime* pseudosquare values are considered,[2] can be used to show that there exist primes with arbitrarily large least primitive roots.

Numeric results for pseudosquares have also been used by Wedeniwski ([Wed01]) to improve the upper bound for the least quadratic nonresidue of a squarefree natural number, $n$, and by Bach and Huelsbergen ([BH93]), to support a heuristic argument on the smallest $x$ generating the multiplicative group modulo $n$.

[2]Western and Miller, for example, tabulated such values in [WM68] pp. xv.

A natural application of pseudosquares is the problem of perfect square recognition. Cobham, in a 1966 IBM Technical Report ([Cob66]), developed an efficient algorithm for determining whether an integer $N$ is a perfect square based on the growth of least quadratic nonresidues. A similar result was given by Bach and Sorenson in [BS93]. Indeed, the results of Williams *et al.*[Wil98] show that to determine whether a single-precision integer is a perfect square, it is sufficient to examine, for example, a 32-bit integer modulo the primes $q \leq 101$, or a 64-bit integer modulo the primes $q \leq 277$. A variant of this idea is used in the popular GNU Multi-Precision (GMP) programming library ([AB]).

### 5.1.2 Pseudosquares and Primality Testing

Perhaps the most interesting application of pseudosquares is in the area of primality testing. In [LPW96], Lukes *et al.* indicated that a sufficiently rapid growth rate of pseudosquares would lead to a deterministic polynomial-time algorithm for determining the prime character of an integer $N$. At the time, the best known unconditional result for proving primality was due to Adleman, Pomerance, and Rumely ([APR83]), and offered a time complexity of $O((\log N)^{c \log\log\log N})$. In August 2002, Agrawal, Kayal,and Saxena ([AKS02]) described an unconditional, deterministic algorithm for proving primality with time complexity $O((\log N)^{12+o(1)})$. This result was later improved by Lenstra and Pomerance (described by Bernstein in [Ber02]) to $O((\log N)^{6+o(1)})$.

This trend raises the obvious question: "how far can the time complexity of unconditional, deterministic primality proving be improved"? In Section 5.2.2, numerical evidence for a conjecture on this point will be offered. First, the issue of how pseudosquares growth is related to the problem of determining the prime character of an integer will be examined.

In the aforementioned [Hal33], Hall was the first to demonstrate a primality test involving the pseudosquares. This test was based on a formalization of some fallacious ideas ([Leh30]) originally put forth by Seelhoff in [See86], and later espoused by Cole [Col03][3] and Kraitchik [Kra29].

The main idea of this test involved what Hall termed *apparent residues*[4], defined as follows.

**Definition 5.2** Apparent Residues and Nonresidues

*Let $N \in \mathbb{Z}$ be odd, and $p, q \in \mathbb{Z}$ be odd primes. Furthermore, define*

$$p' = \left(\frac{-1}{p}\right) p = (-1)^{\frac{p-1}{2}} p$$

*If* $\left(\frac{N}{p}\right) = 1$, *then* $p'$ *is said to be an* apparent residue *of* $N$. *If* $\left(\frac{N}{q}\right) = -1$ *then* $q' = (-1)^{\frac{q-1}{2}} q$ *is said to be an* apparent non-residue *of* $N$.

The apparent residue character of 2 and $-1$ were defined in a similar manner. Hall then proved the following theorem.

**Theorem 5.2** *If all the (not necessarily prime) factors of $N$ are less than $L_p$, and if the primes $-1, 2, -3, 5, \ldots, (-1)^{\frac{p-1}{2}} p$ can be divided into two classes: the apparent residues of $N$, $\mathcal{A} = \{a_1, a_2, \ldots, a_s\}$ and the apparent nonresidues of $N$,*

[3]In an editorial in *Notices of the* AMS ([Kna99]), the story is told of Cole's 1903 address to the Society, entitled *On the Factoring of Large Integers.* It is said the lecture was "met with a standing ovation after he lectured without saying a single word, multiplying two large integers and verifying that their product was $2^{67} - 1$." The description given in [Col03] is somewhat more verbose than this, however, and includes the discussion of Seelhoff's idea.

[4]This was a translation of Kraitchik's "résidues éventuelles."

*$\mathcal{B} = \{b_1, b_2, \ldots, b_r\}$, such that every member $a_i \in \mathcal{A}$ is a true quadratic residue of $N$, and the product of every pair of elements $b_i b_j \in \mathcal{B}$ is also a true quadratic residue of $N$ then $N$ is either a prime or a power of a prime.* ■

Though actually using this method as a primality test is difficult,[5] Beeger successfully used it twice, first in [Bee39] to prove the 12-digit cofactor of $2577687858367 = 17 \cdot 151628697551$ prime,[6] then in [Bee46] to prove a 13-digit factor of $12^{45}+1$ prime.

Dan Shanks, in correspondence with D. H. Lehmer ([Wil98]) implied a different test involving the pseudosquares when he noted the following theorem:

**Theorem 5.3** *If $N \equiv -1 \pmod 4$ is a base-$q$ probable prime, that is, if $q^{N-1} \equiv 1 \pmod N$ for all primes $q \leq p$, then any prime divisor $P \equiv 1 \pmod 4$ of $N$ must satisfy $P \geq L_p$.* ■

The implied test is as follows: if it is known that $N \equiv -1 \pmod 4$ is the product of at most two primes, and if $N < L_p$, then $N$ is a prime if $q^{N-1} \equiv 1 \pmod N$ for all primes $q \leq p$. Unfortunately, like Hall's test, this algorithm is of limited practical use. Selfridge and Weinberger [Wil78], however, extended this idea and their extension, with modifications by Williams in [LPW96], became the first practical primality test to use pseudosquares. This test will now be introduced via a trio of lemmas.

**Lemma 5.1** *Let $s \geq 1$ be the value for which $2^s \| m$,* i.e.$2^s \mid m$, *but $2^{s+1} \nmid m$. If*

$$b^{\frac{m}{2}} \equiv -1 \pmod N$$

[5]To say the least. If $N$ is not known to be prime and the Jacobi symbol $\left(\frac{m}{N}\right) = 1$, the problem of determining whether a given integer $m$ is a quadratic residue modulo $N$ is believed to be as difficult as the problem of factoring integers ([MvOV96], pp. 99).

[6]This is the numerator of the 34th Bernoulli number.

*then any prime factor $p \mid N$ must have the form $p \equiv 1 \pmod{2^s}$.*

**Proof:** Let $p$ be any prime divisor of $N$. Let $\omega = ord_p(b)$, *i.e.*the least positive integer that satisfies $b^{\omega} \equiv 1 \pmod p$. Since $b^{\frac{m}{2}} \equiv -1 \pmod N$, $b^m \equiv 1 \pmod N$ and since $p \mid N$, $b^m \equiv 1 \pmod p$. Thus $\omega \mid m$, but $\omega \not| \frac{m}{2}$, and hence, $2^s \mid\mid \omega$.

Now, since $p$ is prime, $b^{p-1} \equiv 1 \pmod p$, and hence $\omega \mid (p-1)$. Then $2^s \mid (p-1)$, and it follows by definition that $p \equiv 1 \pmod{2^s}$. ∎

**Lemma 5.2** *Given $N \in \mathbb{Z}$ odd, choose $s$ such that $2^s \mid\mid (N-1)$. Suppose $\exists c \in \mathbb{Z}$ such that*

$$c^{\frac{N-1}{2}} \equiv \pm 1 \pmod N$$

*If $\left(\frac{c}{q}\right) = -1$ for some prime factor $q \mid N$ such that $q \equiv 1 \pmod{2^s}$ then $2^s \mid\mid (q-1)$.*

**Proof:** Write $N-1 = 2^s m$ with $m$ odd, and let $\omega = 2^r t = \mathrm{ord}_q(c)$ with $t$ odd. Since $c^{\frac{N-1}{2}} \equiv \pm 1 \pmod N$, and since $q \mid N$, then $c^{N-1} \equiv 1 \pmod q$. Thus $\omega \mid (N-1)$, $2^r t \mid 2^s m$, and since $m$ is odd, $r \leq s$.

It is clear from $\left(\frac{c}{q}\right) = -1$ that $c^{\frac{q-1}{2}} \equiv -1 \pmod q$. Thus $c^{q-1} \equiv 1 \pmod q$, and $\omega = 2^r t \mid (q-1)$ but $\omega = 2^r t$ does not divide $\frac{q-1}{2}$. Thus $2^r \mid\mid (q-1)$. If $q \equiv 1 \pmod{2^s}$ then $2^s \mid q-1$ and $s \leq r$.

Hence, $s = r$, and $2^s \mid\mid (q-1)$. ∎

**Lemma 5.3** *Given $N \in \mathbb{Z}$ odd, choose $s$ such that $2^s \mid\mid (N-1)$. If $2^s \mid\mid (q_1 - 1)$, $2^s \mid\mid (q_2 - 1)$, where $q_1, q_2$ are primes and $c$ is some integer such that $\left(\frac{c}{q_1 q_2}\right) = -1$ then*

$$c^{\frac{N-1}{2}} \not\equiv \pm 1 \pmod{q_1 q_2}$$

**Proof:** $\left(\frac{c}{q_1q_2}\right) = \left(\frac{c}{q_1}\right)\left(\frac{c}{q_2}\right)$ so without loss of generality, choose $\left(\frac{c}{q_1}\right) = 1$ and $\left(\frac{c}{q_2}\right) = -1$. Now let $\omega_1, \omega_2$ be the multiplicative orders of c modulo $q_1$ and $q_2$ respectively.

Suppose:

$$c^{\frac{N-1}{2}} \equiv \pm 1 \pmod{q_1q_2} \tag{5.1}$$

From $\left(\frac{c}{q_2}\right) = -1$ and $2^s \,||\, q_2-1$, it is clear that $c^{\frac{q_2-1}{2}} \equiv -1 \pmod{q_2}$ and $2^s \,||\, \omega_2$. From 5.1, $c^{N-1} \equiv 1 \pmod{q_2}$. Since $2^s \,||\, (N-1)$, $\omega_2$ does not divide $\frac{N-1}{2}$, and

$$c^{\frac{N-1}{2}} \equiv -1 \pmod{q_2} \tag{5.2}$$

Now $\left(\frac{c}{q_1}\right) = 1$, $c^{\frac{q_1-1}{2}} \equiv 1 \pmod{q_1}$, and $\omega_1 \mid \frac{q_1-1}{2} = 2^{s-1}t$ with $t$ odd. Write $\omega_1 = 2^r m$ ($m$ odd), and notice that $r \le s-1$.

Since $c^{N-1} \equiv 1 \pmod{q_1}$, $\omega_1 \mid (N-1)$. Recall that $2^s \,||\, (N-1)$; hence $\omega_1 \mid \frac{N-1}{2}$. So

$$c^{\frac{N-1}{2}} \equiv 1 \pmod{q_1} \tag{5.3}$$

Thus, from 5.2 and 5.3, a contradiction to 5.1 is obtained, and hence

$$c^{\frac{N-1}{2}} \not\equiv \pm 1 \pmod{q_1q_2}$$

■

An efficient primality test involving the pseudosquares may now be given.

**Theorem 5.4** *If*

1. *All prime divisors $q|N$ exceed the bound $B \in \mathbb{Z}^+$,*

*2.* $\frac{N}{B} < L_p$ *for some prime, p,*

*3.* $p_i^{\frac{N-1}{2}} \equiv \pm 1 \pmod{N}$ *for all primes* $p_i$, $2 \leq p_i \leq p$,

*4.* $p_j^{\frac{N-1}{2}} \equiv -1 \pmod{N}$ *for some odd* $p_j \leq p$ *when* $N \equiv 1 \pmod{8}$, *or*

$2^{\frac{N-1}{2}} \equiv -1 \pmod{N}$ *when* $N \equiv 5 \pmod{8}$

*then N is a prime or a power of a prime.*

**Proof:** Suppose an integer $N$ passes the conditions 1–4, but $N$ is not a prime or a prime power. Then $N$ possesses at least 2 distinct prime divisors. Let $q$ be one of these distinct prime divisors. If $N \equiv 3 \pmod{4}$ then $(-1)^{\frac{N-1}{2}} \equiv -1 \pmod{N}$. If $N \equiv 1 \pmod{4}$, then by condition 4 of the theorem, there exists a $b$ such that

$$b^{\frac{N-1}{2}} \equiv -1 \pmod{N}.$$

If we choose $s \in \mathbb{Z}^+$ such that $2^s \mid\mid (N-1)$ then by Lemma 5.1, all prime divisors $q$ of $N$ are of the form $q \equiv 1 \pmod{2^s}$. Furthermore, consider three cases:

*Case 1:* $q \equiv 1 \pmod{8}$ — By condition 1, every prime divisor of $N$ exceeds $B$ and hence $q < \frac{N}{B}$. By condition 2, $q < L_p$ so by the definition of the pseudosquares, there must exist some $p_j < p$ such that $\left(\frac{q}{p_j}\right) = -1$. By Quadratic Reciprocity, $\left(\frac{p_j}{q}\right) = -1$. Notice also that by condition 3, $p_j^{\frac{N-1}{2}} \equiv \pm 1 \pmod{N}$. Thus Lemma 5.2 applies, and $2^s \mid\mid (q-1)$.

*Case 2:* $q \equiv 5 \pmod{8}$ — By the properties of the Legendre symbol, $\left(\frac{2}{q}\right) = -1$. From condition 3, $2^{\frac{N-1}{2}} \equiv \pm 1 \pmod{N}$. Thus Lemma 5.2 applies, and $2^s \mid\mid (q-1)$.

*Case 3*: $q \equiv -1 \pmod{4}$ — By the properties of the Legendre symbol, $\left(\frac{-1}{q}\right) = -1$. Clearly, $(-1)^{\frac{N-1}{2}} \equiv \pm 1 \pmod{N}$, so by Lemma 5.2, $2^s \mid\mid (q-1)$.

Since all prime divisors, $q_i$ of $N$ are of the form $q_i = 1 + 2^s t_i$ for some $t_i$,

$$N = 1 + 2^s t = \prod_{i=1}^{r} 1 + 2^s t_i \quad \text{with } t, t_i \text{ odd.}$$

By a simple parity argument, it is clear that $r$ must be odd. *i.e.*

$$\begin{aligned} 1 + 2^s t &= \prod_{i=1}^{r} 1 + 2^s t_i \\ 1 + 2^s t &= (1 + 2^s t_1)(1 + 2^s t_2) \cdots (1 + 2^s t_r) \\ 1 + 2^s t &\equiv 1 + 2^s \sum_{i=1}^{r} t_i \pmod{2^{s+1}} \end{aligned}$$

And thus $t \equiv \sum_{i=1}^{r} t_i \pmod 2$. Since $N$ is not a prime or a prime power, there must be two distinct primes, $q_1, q_2$ such that $q_1 q_2 \mid N$. By a similar argument as before it may be shown that:

- If $q_1 q_2 \equiv 1 \pmod 8$, there exists a prime $p_k < p$ such that $\left(\frac{q_1 q_2}{p_k}\right) = \left(\frac{p_k}{q_1 q_2}\right) = -1$, as $q_1 q_2 < \frac{N}{B} < L_p$ and $q_1 q_2$ is not a perfect square.
- If $q_1 q_2 \equiv 5 \pmod 8$, then by the properties of the Jacobi symbol $\left(\frac{2}{q_1 q_2}\right) = -1$.
- If $q_1 q_2 \equiv 3 \pmod 4$ then the Jacobi symbol $\left(\frac{-1}{q_1 q_2}\right) = -1$.

Condition 3 says $p_i^{\frac{N-1}{2}} \equiv \pm 1 \pmod N$, so $p_i^{\frac{N-1}{2}} \equiv \pm 1 \pmod{q_1 q_2}$ for all primes $2 \le p_i \le p$, but by Lemma 5.3, $p_i^{\frac{N-1}{2}} \not\equiv \pm 1 \pmod{q_1 q_2}$, a contradiction.

So $N$ must be a prime, or a prime power. ∎

Notice that if $N$ is a nonsquare, and $N < L_p$, then there exists a prime $q$ such that $2 \le q \le p$, and $\left(\frac{q}{N}\right) \ne 1$. Furthermore, if $N$ is prime, conditions 3 and 4 of Theorem 5.4 always hold.

The main consequence of this result is that if $L_p$ grows sufficiently quickly *i.e.*if $p < c(\log L_p)^k$ for fixed constants $c, k$, then Theorem 5.4 offers an unconditional, deterministic polynomial-time primality test.

The growth rate of the pseudosquares will now be examined.

### 5.1.3 Pseudosquare Growth

In [Bac90], Bach gave the following result, conditional upon the Extended Riemann Hypothesis (ERH):

**Theorem 5.5** *Let $\mathcal{G}$ be a nontrivial subgroup of $(\mathbb{Z}/m\mathbb{Z})^*$ (the group of reduced residues modulo $m$) such that $n \in \mathcal{G}$ for all positive $n < x$. Then $x < 2(\log m)^2$.* ■

This result may be used to bound $L_p$ as follows. Consider the subgroup $\mathcal{G}$ of $(\mathbb{Z}/L_p\mathbb{Z})^*$ consisting of all $g$ such that $\left(\frac{g}{L_p}\right) = 1$. Since $L_p$ is a nonsquare, there must be an odd prime $q$ such that $q^\alpha \mid\mid L_p$ with $\alpha$ odd. Let $t$ be a quadratic nonresidue of $q$, *i.e.*$\left(\frac{t}{q}\right) = -1$, and set:

$$\begin{aligned} r &\equiv t \pmod{q^\alpha} \\ r &\equiv 1 \pmod{\frac{L_p}{q^\alpha}} \end{aligned}$$

By the CRT, $r \in (\mathbb{Z}/L_p\mathbb{Z})^*$ and by the properties of the Jacobi symbol, $\left(\frac{r}{L_p}\right) = -1$. Thus, $\mathcal{G}$ is a nontrivial subgroup of $\mathbb{Z}/(L_p)^*$. Also, $n \in \mathcal{G}$ for all $0 < n < p$. By Theorem 5.5, $p < 2(\log L_p)^2$, and hence

$$\log L_p > \sqrt{\frac{p}{2}} \tag{5.4}$$

In [Sch97], Schinzel refines the bounds on $L_p$ to:[7]

$$(1-\epsilon)\sqrt{p} < \log L_p < (2\log 2 + \epsilon)\frac{p}{\log p}$$

for any $\epsilon > 0$ with $p > p_0(\epsilon)$.

Thus, under the conditions of the ERH, Theorem 5.4 offers a deterministic polynomial-time primality test.

Lukes offers an alternate prediction for the growth rate of $L_p$ in [Luk95], pp. 111, based on a density argument and the Prime Number Theorem.[8] Under the stated assumptions, $L_p$ would have a growth rate of the form $2^{(p/\log p)(1+o(1))}$. In other words:

$$\log L_p \approx \frac{p \cdot \log 2}{\log p} \tag{5.5}$$

It should be noted that by the Prime Number Theorem ([HW79] pp. 9), $n \sim \frac{p}{\log p}$ where $p$ is the nth prime, and hence, $\log L_p \approx n \cdot \log 2\,(1 + o(1))$.

This coincides with the assumption given by Bach and Huelsbergen in [BH93] that the pseudosquares provide extreme values of $G(p)$, where $G(n)$ represents the smallest value of $x$ such that the primes $\leq n$ generate the multiplicative group $\mathbb{Z}_n^*$. This leads to the relationship $p \approx \frac{\log L_p \log\log L_p}{log 2}$, and hence, Equation 5.5 [LPW96].

If these predictions hold, then primality proving may be done (via Theorem 5.4) using $O((logN)^{1+o(1)})$ modular exponentiations. Since performing modular exponentiation (using, for instance, the techniques of Schönhage and Strassen [SS71]) incurs a complexity of $O((logN)^{2+o(1)})$, it may therefore be conjectured that the primality of an arbitrary integer $N$ may be proved with $O((logN)^{3+o(1)})$ operations.

[7]Assuming the Extended Riemann Hypothesis (ERH). In the same paper, an unconditional result is also given.

[8]Essentially, assume solutions for $L_p$ are equidistributed in the range $0 < x < 8p_2p_3\cdots p_n$, and hence $L_p \approx \frac{8p_2p_3\cdots p_n}{\prod_{i=1}^{n}(p_i-1)/2}$, so by Merten's Theorem [HW79], and the Prime Number Theorem, $L_p \approx c2^n \log p$ for $c = 2e^\gamma$ where $\gamma = 0.57721$ is Euler's constant.

### 5.1.4 Applications to Cryptography

In 2003, Bernstein [Ber03a] presented a fast, secure public-key signature scheme based on the Rabin-Williams [Ber03b] cryptosystem. In this system, a private key is a pair $(p, q)$, with $p, q$ prime, $p \equiv 3 \pmod 8$ and $q \equiv 7 \pmod 8$. The public key is the product $N = pq$. The signature scheme is defined as follows.

**Definition 5.3** *Given a message, $\mathcal{M}$, a publicly known hash function, $H$,*[9] *and a public key $N = pq$, a* standard signature *is defined to be a vector $(e, f, r, s)$ such that $e \in \{1, -1\}$, $f \in \{1, 2\}$, $r$ is a random bitstring of length $B$, and $s \in \mathbb{Z}$ satisfying:*

$$fs^2 \equiv eH(r||\mathcal{M}) \pmod N$$

where $||$ denotes bitwise concatenation.

The difficulty of forging a Rabin-Williams signature is based on the problem of determining a square root modulo the composite integer $N$. It can be shown, for instance in [Wil80], that the difficulty of this problem is equivalent to the problem of factoring $N$.

Verification of Rabin-Williams signatures is extremely fast, requiring only a single modular squaring. Bernstein noted, however, that verification can be made even faster by transmitting what he calls an *expanded signature*, and replacing this modular squaring with a randomized test.

**Definition 5.4** *Given the message $\mathcal{M}$ and its standard signature, $(e, f, r, s)$, define an* expanded signature *as the vector $(e, f, r, s, t)$ where $t$ is defined as the integer*

[9]A hash function is a function mapping an arbitrary-length bitstring into a fixed-length bitstring. Cryptographic hash functions are typically chosen such that it is computationally infeasible to find two distinct bitstrings $x_1, x_2$ that evaluate to the same hash value, $H(x_1) = H(x_2)$, and that given $y$, it is computationally infeasible to determine its pre-image $x$, such that $H(x) = y$.

*satisfying:*

$$fs^2 - eH(r||\mathcal{M}) - tN = 0$$

Clearly, verification of an expanded signature may be achieved by evaluating $c = fs^2 - eH(r||\mathcal{M}) - tN$ and testing whether $c = 0$. A faster, randomized verification scheme is obtained by mapping it to a random quotient ring, for example, by generating a random, secret 100-bit prime, $P$, setting

$$\begin{aligned} s' &\equiv s \pmod{P} \\ t' &\equiv t \pmod{P} \\ N' &\equiv N \pmod{P} \\ h' &\equiv H(r||\mathcal{M}) \pmod{P} \end{aligned}$$

and computing $c' \equiv fs'^2 - t'N' - eh' \pmod{p}$. Note that this result is zero if $c = 0$, and is virtually guaranteed[10] to be nonzero if $c \neq 0$.

Bernstein further noted that the pseudosquare test of Theorem 5.4 offers the fastest known method for verifying whether a candidate prime for this verification scheme ($P$) is in fact a provable prime. *i.e.*If it can be shown that $P$ has no prime divisors $< 2^{20}$, then (as illustrated in Table 5.2) $\frac{N}{B} < L_{367}$, and a proof of primality requires only 73 modular exponentiations.

### 5.1.5 Applications to Networking — Spam Prevention

The unmetred nature of Internet mail delivery has resulted in a proliferation of Unsolicited Commercial Email—typically referred to as UCE or *spam*. Despite an in-

[10]With probability $1 - 2^{-100}$.

credibly low response rate, purveyors of spam have been able to maintain profitability due to the essentially zero-cost nature of electronic mail delivery.

One oft-cited remedy for the spam issue is the adoption of a pay-per-send pricing model for electronic mail. In 1992, Dwork and Naor [DN92] presented an innovative computational technique for implementing a pay-per-send infrastructure. Under their proposal, the sender of a electronic mail message was first required to perform a message-specific computation requiring a moderate investment[11] of CPU time. Before accepting a message for delivery, the mail gateway would verify this computation; accepting or rejecting the message accordingly. By design, verification of this computuation required significantly less effort than was needed to produce it in the first place—often by several orders of magnitude.

Under the Dwork and Naor proposal, the computational cost of sending millions of messages a day could be made prohibitively expensive, shifting the computational workload from sender to receiver without imposing a significant burden on regular users of the electronic mail infrastructure.

Variants of this idea, sometimes referred to as the *proof of work* (POW) concept, have been demonstrated to be effective in a variety of resource exhaustion application, including Adam Back's HashCash proposal [Bac97], Rabin, Shamir, and Wagner's [RSW96] cryptographic time-lock, and Juels and Brainard's Client Puzzles [JB99].

## Pseudosquares and Spam Prevention

A variant of modular square root problem may be adapted to implement a workable anti-spam solution for electronic mail that involves the pseudosquares. The technique

[11]Typically, 2-10 seconds on of CPU time on a desktop workstation.

works as follows.

First, a mail server generates an extremely large prime $p \equiv -1 \pmod 4$ and a public hash function, $H$. Once chosen, these parameters may remain the same for the remainder of the protocol.

Before accepting a message $\mathcal{M}$ for queuing, the mail server issues a challenge consisting of the large prime $p$ and an R-bit random bitstring, $r$. Using this information, the sending client computes the message hash

$$h = \epsilon H(r||\mathcal{M})$$

and finds a modular square root

$$s^2 \equiv h \pmod p,$$

where $\epsilon \in \{-1, 1\}$ is chosen to ensure that the Legendre symbol, $\left(\frac{h}{p}\right) = 1$. The client then sends the 4-tuple $(s, \epsilon, k, \mathcal{M})$, where $k$ is the solution to $s^2 - h - kp = 0$ to the server for verification.

At this point, the server may verify the computation by evaluating whether the solution to $c = s^2 - h - kp$ is exactly zero. An even faster verification is possible using the randomized verification idea of Section 5.1.4, where the server chooses a random 100-bit prime, $Q$, verifies its primality using the pseudosquare test of Theorem 5.4, and reduces the verification parameters modulo this prime; *i.e.*,

$$\begin{aligned}
\tilde{s} &\equiv s^2 \pmod Q \\
\tilde{k} &\equiv k \pmod Q \\
\tilde{p} &\equiv p \pmod Q \\
\tilde{h} &\equiv \epsilon H(r||\mathcal{M}) \pmod Q.
\end{aligned}$$

Verification of the problem is a matter of evaluating whether $\tilde{s} - \tilde{h} - \tilde{k}\tilde{p} \equiv 0 \pmod{Q}$. Note that this result is zero whenever $c = 0$, and is virtually guaranteed[12] to be nonzero if $c \neq 0$.

Note that the value $\tilde{p}$ may be precomputed whenever a new random prime, $Q$, is chosen. A new prime, $Q$, should be generated at regular intervals.[13]

The size of the prime $p$ may be varied to change the workload demanded of remote clients. Since $p \equiv -1 \pmod 4$, the most efficient means known for computing the modular square root, $s^2 \equiv h \pmod p$, is to evaluate $s \equiv h^{\frac{p+1}{4}} \pmod p$. This modular exponentiation has an expected runtime of $O((\log p)^3)$ bit operations using the traditional algorithm [MvOV96]. Thus, the computational effort expended by the client is effectively parameterized by the size of the prime $p$.

One consequence of this parameterization is that untrusted (or offending) clients can be asked to perform a more intensive computation than known (or trusted) clients; an idea which leads naturally into the concept of a decentralized web-of-trust [14] for electronic mail delivery.

## 5.2 Pseudosquare Results

In [Luk95], Lukes offered empirical evidence to support both Bach's estimate ($\log L_p > \sqrt{\frac{p}{2}}$), and the Bach and Huelsbergen density prediction ($\log L_p \approx \frac{p \log 2}{\log p}$) by computing the pseudosquares for all primes $q \leq 277$, and comparing the results to 5.4 and

[12]With probability $1 - 2^{-100}$.

[13]Typically, a new prime $Q$ would be chosen after either a set amount of traffic had been processed, or a fixed time interval had elapsed, whichever occurred first.

[14]A model which was originally popularized by the Pretty Good Privacy (PGP) encryption package [Sta95].

5.5. Using CASSIE, the table of pseudosquares was extended to include all primes $q \leq 359$ and it was shown that the predictions of 5.4 and 5.5 still hold. These results are given in Section 5.2.2.

### 5.2.1 Construction of the Problem

To extend the table of pseudosquares, two separate computations were performed. Both computations were performed in software, and used Bernstein's doubly-focused enumeration optimization ([Ber04]) to speed the computation.

#### First Run

The first computation, a proof of concept, was a doubly-focused enumeration implemented in software over two processors. The underlying hardware was a dual-processor Athlon MP 2000+. The software was compiled using GCC 2.96 under Red Hat Linux 7.3 (kernel 2.4.18-27.7), using the -O2 optimization.

To partition the problem over two processors, the acceptable pseudosquare residues for 3, 5, and 8 were combined to produce the congruence condition $x \pmod{120} \in \{1, 49\}$. Each of these two possibilities was then converted to a normalization function, $x = 1 + 120y$, and $x = 49 + 120y$, and assigned to its own processor. Apart from the normalization function the problem setup on each of the processors was identical.

The problem setup was very similar to Bernstein's in [Ber04]. The primes from 7 to 73 were used as doubly-focused moduli, arranged in the following manner: $\mathcal{M}_n = \{7, 13, 29, 31, 71, 41, 43, 59, 61\}$ and $\mathcal{M}_p = \{11, 17, 19, 23, 73, 37, 47, 53, 67\}$.

Thus, the sieve searched for solutions $x = t_p M_n - t_n M_p$, where $M_n = \prod_{m_n \in \mathcal{M}_n} m_n =$

$36,854,760,367,243$, and $M_p = \prod_{m_p \in \mathcal{M}_p} m_p = 36,838,009,702,043$.

Values emerging from the doubly-focused sieve were further filtered by passing them through an exclusion sieve containing the primes 79–127. Remaining results were filtered to remove the perfect squares, and then tested against the primes up to 400 to determine where their pseudosquare behaviour broke down.

The run was completed on April 6, 2003. The process took approximately 298 hours, giving an effective canvass rate for the software sieve of $2,063,394,191,690,106 \approx 2.06 \times 10^{15}$ trials per second. The sieve, by virtue of the normalization function, searched all solutions up to $120 \times 2^{64}$, and in addition to verifying the previous results up to $L_{281}$, was able to find 6 previously unknown pseudosquare values: $L_{293}$ to $L_{317}$. These results are summarized in Table 5.2.

## Second Run

Once CASSIE had proved successful in the initial run, the pseudosquare computation was retooled for implementation in software over 180 processors. The underlying hardware was the University of Calgary's ACL; a Beowulf cluster consisting of 139 dual-Xeon Pentium IV processors running at 2.4 GHz. The software was compiled using GCC 2.96 under Red Hat Linux 7.3 (kernel 2.4.18-27.7), using the -O2 optimization.

To partition the problem over 90 dual-processor units, the acceptable pseudosquare residues for 3, 5, 8, 11, 13, and 17 were combined to produce 180 acceptable residue classes (mod 120120). Each of these residue classes was converted to a normalization function. Using the MPI library [GLS94], the CASSIE software running on each of the ACL nodes was able to determine which normalization function to use.

The rest of the problem setup was identical for each of the processors. The primes from 17 to 83 were arranged into two sets: $\mathcal{M}_n = \{17, 23, 29, 31, 37, 41, 47, 53, 71\}$ and $\mathcal{M}_p = \{19, 43, 59, 61, 67, 73, 79, 83\}$.

The sieve then searched for solutions $x = t_p M_n - t_n M_p$, where $M_n = \prod_{m_n \in \mathcal{M}_n} m_n = 94,309,209,838,733$ $M_p = \prod_{m_p \in \mathcal{M}_p} m_p = 94,298,926,699,921$.

Values emerging from the doubly-focused sieve were further filtered by passing them through an exclusion sieve containing the primes 89–127. Remaining results were filtered to remove the perfect squares, and then tested against the primes up to 400 to determine where their pseudosquare behaviour broke down.

The run was completed on July 26, 2003. The process took approximately 585 hours, giving an effective canvass rate for the software sieve of:

$$1,052,147,624,944,915,166 \approx 1.05 \times 10^{18}$$

trials per second. The sieve, by virtue of the normalization function, searched all solutions up to $120120 \times 2^{64}$, and in addition to verifying the previous results up to $L_{317}$, was able to find 6 previously unknown pseudosquare values: $L_{331}$ to $L_{359}$. These results are summarized in Table 5.2.

### 5.2.2 Numerical Results

In Figure 5.1, the values of $\log L_p$ obtained from CASSIE have been plotted against $p$, and compared with the experimental predictions, $L_p > e^{\sqrt{p/2}}$, and $\log L_p \approx \frac{p \log 2}{\log p}$. In Figure 5.2, pseudosquare growth is shown as a function of $n$, where $p_n$ is the $n^{th}$ prime. The straight line represents the least squares line fitted to this data, and is

| $p$ | $L_p$ | Source |
|---|---|---|
| 3 | 73 | Kraitchik (1924) |
| 5 | 241 | *Movable strips* |
| 7 | 1 009 | |
| 11 | 2 641 | |
| 13 | 8 089 | |
| 17 | 18 001 | |
| 19 | 53 881 | |
| 23 | 87 481 | |
| 29 | 117 049 | |
| 31 | 515 761 | |
| 37 | 1 083 289 | |
| 41 | 3 206 641 | |
| 43 | 3 818 929 | |
| 47 | 9 257 329 | |
| 53 | 22 000 801 | Lehmer (1928) |
| 59, 61 | 48 473 881 | *Bicycle chain sieve* |
| 67 | 175 244 281 | Lehmer (1954) |
| 71, 73 | 427 733 329 | SWAC |
| 79 | 898 716 289 | |
| 83, 89, 97 | 2805 544 681 | Lehmer, Lehmer, and Shanks (1970) |
| 101 | 10 310 263 441 | *DLS-127* |
| 103 | 23 616 331 489 | |
| 107,109 | 85 157 610 409 | |
| 113,127 | 196 265 095 009 | |
| 131,137,139 | 2 871 842 842 801 | Lehmer (1973) |
| 149,151 | 26 250 887 023 729 | *DLS-157* |
| 157, 163, 167 | 112 434 732 901 969 | Patterson, Williams (1988) |
| 173,179 | 178 936 222 537 081 | *UMSU* |
| 181,191 | 696 161 110 209 049 | |
| 193 | 2 854 909 648 103 881 | Stephens, Williams (1989) |
| 197,199 | 6 450 045 516 630 769 | *OASiS* |
| 211,223 | 11 641 399 247 947 921 | |
| 227 | 190 621 428 905 186 449 | Lukes, Williams (1991) |
| 229 | 196 640 248 121 928 601 | *OASiS II* |
| 233 | 712 624 335 095 093 521 | Patterson, Williams (1994) |
| 239 | 1 773 855 791 877 850 321 | *MSSU* |
| 241 | 2 327 687 064 124 474 441 | |
| 251 | 6 384 991 873 059 836 689 | |
| 257 | 8 019 204 661 305 419 761 | |
| 263,269,271 | 10 198 100 582 046 287 689 | |
| 277 | 69 848 288 320 900 186 969 | |
| 281 | 208 936 365 799 044 975 961 | Bernstein (2001) |

*Table 5.1*: Previous Pseudosquare Results

| $p$ | $L_p$ | Source |
|---|---|---|
| 283 | 533 552 663 339 828 203 681 | Williams, Wooding (2003) |
| 293,307 | 936 664 079 266 714 697 089 | *CASSIE* |
| 311,313,317 | 2 142 202 860 370 269 916 129 | |
| 331 | 13 649 154 491 558 298 803 281 | Williams, Wooding (2003) |
| 337 | 34 594 858 801 670 127 778 801 | *CASSIE / ACL* |
| 347, 349 | 99 492 945 930 479 213 334 049 | (180 processors) |
| 353, 359 | 295 363 487 400 900 310 880 401 | |
| 367 | $> 120120 \times 2^{64}$ | |

*Table 5.2*: New Pseudosquare Results

given by:

$$y = 0.671121885x + 4.77028237$$

Even with the relatively small number of data points, the slope of the least squares fit in Figure 5.2 appears to be approaching the predicted value of $\log 2 = 0.69315$, *i.e.*$L_p$ is of the form $2^{n(1+o(1))}$.

It is clear then, that the pseudosquare results obtained to date support the predictions of Equations (5.4) and (5.5). This is at least empirical evidence that the polynomial-time nature of the primality test of Theorem 5.4 holds even in the absence of the ERH.

## 5.3 Minus Class Numbers of Imaginary Cyclic Quartic Fields

Let $p$ be a prime and let $\zeta_p$ be a primitive $p^{th}$ root of unity. Let $N$ denote the maximal imaginary subfield of degree $d$, a power of 2, of the cyclotomic field $\mathbb{Q}(\zeta_p)$ and let $N^+$ denote the real quadratic subfield of degree $d/2$ of $N$. The *minus (relative) class number* $h_N^-$ of $N$ is given by

$$h_N^- = h_N/h_{N^+}$$

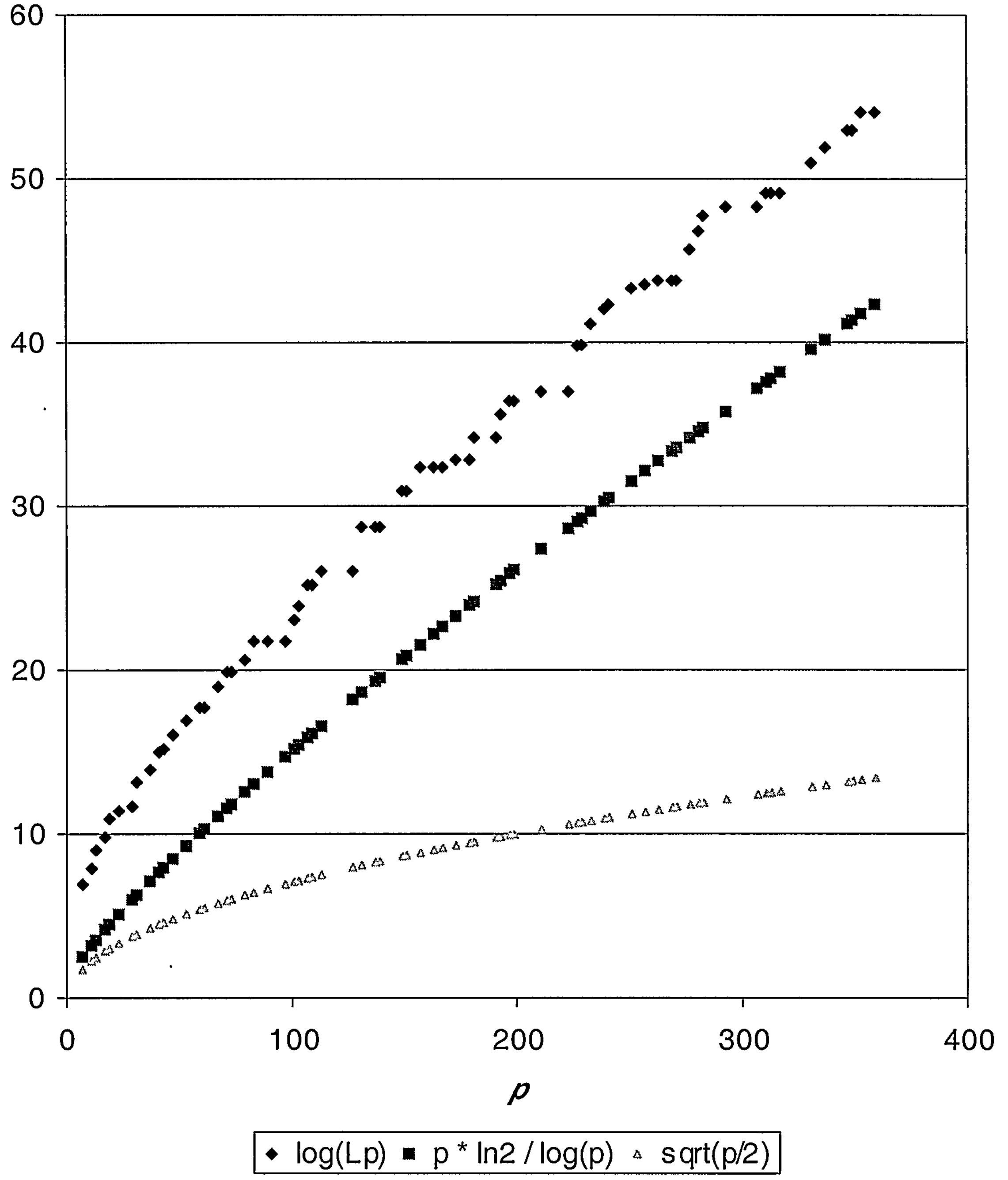

*Figure 5.1*: Pseudosquare growth vs. $p$

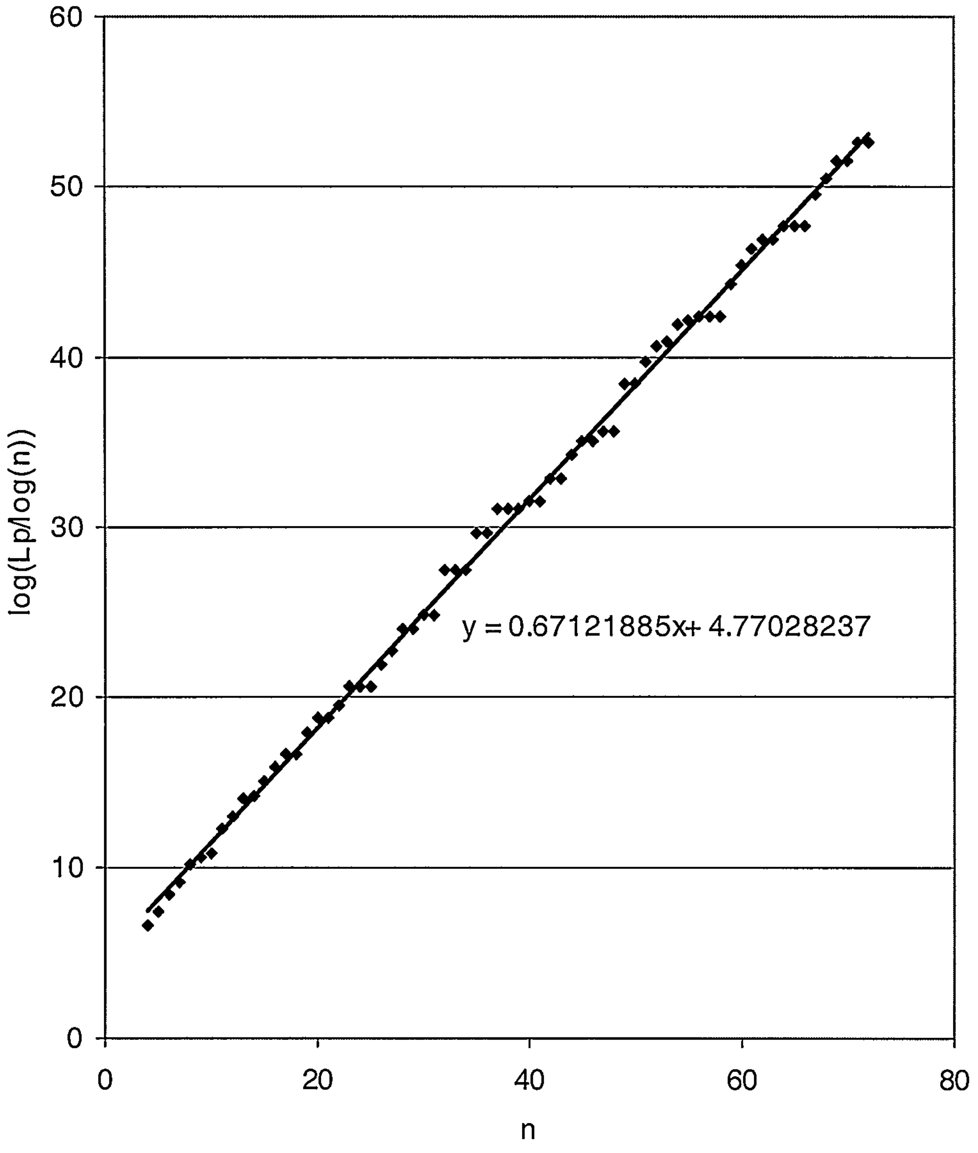

*Figure 5.2*: Pseudosquare growth vs. $n$

where $h_N$ and $h_{N^+}$ are the class numbers of $N$ and $N^+$, respectively. For example, if $p \equiv -1 \pmod 4$, then $N_p := N = \mathbb{Q}(\sqrt{-p})$, $N^+ = \mathbb{Q}$, and $h_p^- := h_N^-$ is the class number of $\mathbb{Q}(\sqrt{-p})$. In this case $h_p^- < p$ is always true.

If $p \equiv 5 \pmod 8$, there exist integers $a$ and $b$ such that $p = a^2 + b^2$, $a \equiv -1 \pmod 4$, $b \equiv 2 \pmod 4$, and $ab \equiv 2 \pmod 8$. Furthermore, these conditions suffice to determine $a$ and $b$ uniquely. In this case

$$N_p := N = \mathbb{Q}\left(\sqrt{-\left(p + b\sqrt{p}\right)}\right), \quad N_p^+ := N^+ = \mathbb{Q}(\sqrt{p}) \ ,$$

and if $h_p^-$ is the minus class number of $N_p$, then $h_p^-$ is not necessarily less than $p$. In fact, for any $c > 0$ it can be shown that there exists an infinitude of values of $p$ such that $h_p^- > cp$. For the remainder of this discussion, we will restrict our attention to the case of $h_p^-$ when $p \equiv 5 \pmod 8$.

Let $\chi_p$ denote the only quartic Dirichlet character of conductor $p$ such that $\chi_p(2) = i$ $(i^2 + 1 = 0)$. If, as usual, the Dirichlet $L$-function is denoted as

$$L(s, \chi_p) = \sum_{n=1}^{\infty} \frac{\chi_p(n)}{n^s} \ ,$$

then

$$h_p^- = \frac{p}{2\pi^2} \left|L(1, \chi_p)\right|^2 = \frac{1}{2} \left|L(0, \chi_p)\right|^2 \ . \tag{5.6}$$

In [Lou98], Louboutin developed a method of approximating the computation of $L(0, \chi_p)$. Using this result, he was able to compute unconditionally all the values of $h_p^-$ for $p < 10^7$. Furthermore, by restricting the value of $p$ such that $\chi_p(q) = 1$ for $q \in \{3, 5, 7, 11, 13, 17, 19, 23, 29\}$, he was able to use his method to discover that if $p = 1679516029$, then $h_p^- = 904595821 > p/2$ ($h_p^-/p \approx 0.5386$), but he was unable to say whether this was the least such $p$ for which $h_p^- > p/2$.

The purpose of this investigation is to attempt to find values of $p$ for which larger values of the ratio $h_p^-/p$ are obtained by prescribing the quartic character $\chi_p(q)$ for the first several odd primes $q$. Specifically, we addressed a challenge by Louboutin to find a $p$ for which $h_p^-/p > 1$. Using a special construction and sieve methods, a conditional result (contingent upon the ERH) was successfully obtained. The construction of this sieve problem will now be described.

### 5.3.1 Tabulation Approach

The main idea is a modification of Louboutin's earlier idea of prescribing the quartic character of the primes for which $h_p^-$ could be evaluated. Instead of using only the first 9 primes, prescribe $p$ as follows:

- $\chi_p(q) = 1$ for $3 \leq q \leq 31$ (the first 10 odd primes),
- $\chi_p(q) = 1$ for 5 of the next 10 primes.

Since

$$L\left(1, \chi_p\right) = \prod_q \left(\frac{q}{q - \chi_p(q)}\right) \quad ,$$

this maximizes the first several terms of the Euler product for $\left|L\left(1, \chi_p\right)\right|$.

In order to filter out possible prime candidates quickly by recognizing whether or not this characterization holds, a result of Emma Lehmer [Leh58] may be employed. For $p = a^2 + b^2$ as above and $\left(\frac{q}{p}\right) = 1$ (clearly if $\chi_p(q) = 1$, $\left(\frac{q}{p}\right) = 1$)

- $\chi_p(q) = \left(\frac{-1}{q}\right)$ if $q \mid b$
- $\chi_p(q) = \left(\frac{-2}{q}\right)$ if $q \mid a$

| $p_i$ | $p = a^2 + b^2$ | $h_p^-/p$ |
|---|---|---|
| 47 | 76400598855755832109 | 0.58 |
| 53 | 137096872440020143225509 | 0.57 |
| 59 | 138095037314298715222509 | 0.68 |
| 61 | 151450834829453613960250 9 | 0.61 |
| 67 | 1514508348294536139602509 | 0.61 |

*Table 5.3*: Minimal $p = a^2 + b^2$ with $a = 3$

- $\chi_p(q) = \left(\frac{-2\lambda(\lambda+1)}{q}\right)$ if $ab \not\equiv 0 \pmod{q}$ and $a \equiv \mu b \pmod{q}$ for any $\mu$ such that $\mu^2 \equiv (\lambda^2 - 1)^{-1} \pmod{q}$, and $\lambda \not\equiv 0, \pm 1 \pmod{q}$.

She also showed that there are exactly $\frac{1}{4}\left(q - 4 - 3\left(\frac{-1}{q}\right) - 2\left(\frac{-2}{q}\right)\right)$ such values of $\mu \bmod q$.

Let $P = p_1 p_2 \cdots p_i$ be the product of the first $i$ odd primes. Our initial sieve-based approach involved fixing the value of $a$ such that $a \equiv 0 \pmod{3}$, $b \equiv 2 \pmod{4}$, $b \equiv \mu^{-1} \pmod{P}$, and finding minimal solutions such that $p = a^2 + b^2$ is prime. This attempt did not produce values for $h_p^-$ much better than to those obtained via a direct calculation, although considerably less processing power was required. For instance, the brute force tabulation of $h_p^-$ for the 113,764,515 primes $p < 10^{10}, p \equiv 5 \pmod{8}$ took 1564 days of CPU time on 269 2.4 GHz Xeon processors running Linux, producing a maximal $h_p^-/p \approx 0.69599$. The sieve problems for the choices $a = 3$ and $a = 9$ took approximately 75 minutes on a single-processor AMD Athlon 2000+, and produced similar ratios. These sieve results are summarized in Tables 5.3 and 5.4. A detailed account of the brute-force tabulation is given in [JWW04].

| $p_i$ | $p = a^2 + b^2$ | $h_p^-/p$ |
|---|---:|---|
| 47 | 514504779123313322581 | 0.54 |
| 53 | 4713675167444727400981 | 0.58 |
| 59 | 6289887916225721232181 | 0.57 |
| 61 | 17359514491514784863352181 | 0.68 |
| 67 | 9189441298229683073700181 | 0.64 |
| 71 | 96831855201472164369820981 | 0.63 |
| 73 | 96831855201472164369820981 | 0.63 |

*Table 5.4*: Minimal $p = a^2 + b^2$ with $a = 9$

### 5.3.2 Construction Approach

The next approach was to construct a value $p$ using a technique of Teske and Williams [TW99]. Here, we obtained an approximation to $|L(1, \chi_p)|^2$ by computing

$$\prod_{q \le Q} \left( \frac{q^2}{q^2 - 1} \right) .$$

(We assume $\chi_p(q) = 1$ for all primes $q \le Q$.) If $Q = 257$, then this quantity exceeds $2\pi^2$; thus, since the tail of the modulus of the Euler product is likely to be near 1, by (5.6) it is reasonable to hope that $h_p^- > p$ for $p$ such that $\chi_p(q) = 1$ for all odd primes $q \le 257$.

To find such values of $p$ we first computed

1. $b \equiv 2 \pmod{4Q_1}$, where $Q_1$ is the product of all primes $q \le 257$ such that $q \equiv 1 \pmod 4$;

2. $a = Q_2$, where $Q_2$ is the product of all primes $q \le 257$ such that $q \equiv 3 \pmod 8$.

By Lehmer's results in Section 5.3.1 it is clear that $\chi_p(q) = 1$ for $q \mid Q_1 Q_2$. For the primes $q \equiv 7 \pmod 8$, we need to find $X$ such that $aX \equiv \mu b \pmod q$, $\mu^2 \equiv (\lambda^2 - 1)^{-1} \pmod q$ and $\left(\frac{\lambda(\lambda+1)}{q}\right) = -1$. Again, we used CASSIE with this sieve construction to produce suitable values of $X$ modulo $Q_3$ where $Q_3$ is the product of all primes $q \leq 257$ with $q \equiv 7 \pmod 8$.

If $(Q_2X)^2 + b^2$ is not a prime, we put $Y = X + tQ_3$ in place of $X$ and put

$$f(t) = Q_2^2Y^2 + b^2 = Q_2^2Q_3^2t^2 + 2Q_2^2Q_3Xt + Q_2^2X^2 + b^2 \ .$$

If we find prime values for $f(t)$, then such primes $p$ will satisfy $\chi_p(q) = 1$ for all $q \leq 257$.

Using $q \leq 257$ yields

$$\begin{aligned} Q_3 &= 87921368557911036603696415 3 \\ Q_2 &= 604551699077399691831592668 3 \\ Q_1 &= 310733977104002095156589937548709682261192960589 0 \end{aligned}$$

The difficulty with this approach is that the resulting values for $p$ are much too large for $h_p^-$ to be evaluated by our techniques in Section 5.3.1. However, an estimation of $h_p^-/p$ may be obtained using a technique originally due to Bach, and later modified by te Riele and Williams [tRW03].

### 5.3.3 Sieve Results

Using the construction in the previous section, CASSIE was able to obtain 14052 values of X in 18 minutes on a dual-processor AMD Athlon 2000+, the smallest of which was 1385546961. Using the values $X = 62204701189$ and $t = 0$, we find that $f(t)$ is a prime

$$p = 9655560452687049830090990196397985846572467001481350068249474236723465700952560829330262763823669 \quad \text{(97 decimal digits).}$$

The estimation technique of [tRW03] notes the following:

If $S(x) = \sum_{j=0}^{x-1}(x+j)\log(x+j)$, $B(x,\chi_p) = \sum_{q<x} \frac{q}{q-\chi_p(q)}$, $a_j = (x+j)\frac{\log(x+j)}{S(x)}$, then under the ERH

$$\left|\log L\left(1,\chi_p\right) - \sum_{j=0}^{T-1} a_j \log B(T+j,\chi_p)\right| < A(T,p) \ ,$$

where

$$A(T,p) = c(p)G(T) + H(T) \ ,$$

$c(p) = \frac{2}{3}(\log p + \frac{5}{3})$, and $G(x)$, $H(x)$ are defined in [tRW03].

Putting

$$S(T,p) = \sum_{i=0}^{T-1} a_i \mathrm{Re}(\log B(T+j,\chi_p)) = \sum_{q\le 2T-1} w(q)\log\left|\frac{q}{q-\chi_p(q)}\right| \ ,$$

where

$$w(q) = \begin{cases} 1 & \text{for } q < T \\ \sum_{j=q-T+1}^{T-1} a_j & \text{for } T \le q < 2T-1 \ , \end{cases}$$

then

$$\left|\log\left|L\left(1,\chi_p\right)\right| - S(T,p)\right| < A(T,p) \ .$$

Hence

$$\log\left|L\left(1,\chi_p\right)\right| > S(T,p) - A(T,p) \ . \tag{5.7}$$

Using this method with our 97 digit prime and $T = 16830000$ we obtain:

$$S(T,p) = 1.50800717, \quad A(T,p) = 0.01499807 \ .$$

Thus, by (5.7)

$$\begin{aligned}\log|L(1,\chi_p)| &> 1.50800717 - 0.01499807 \\ &> 1.49300910 \\ &> \log(\sqrt{2\pi^2}) \quad (\approx 1.491303476)\end{aligned}$$

and hence, by (5.6)

$$h_p^-/p = \frac{|L(1,\chi_p)|^2}{2\pi^2} > 1$$

as desired.

# Chapter 6

# Conclusions and Summary

*Say what you have to say and the first time you come to a sentence with a grammatical ending; sit down.*

—Winston Churchill

## 6.1 Summary

In this thesis, the development of a hybrid hardware/software sieve named CASSIE has been detailed. CASSIE represents a deviation from previous fixed-plus-variable sieve designs in that a dual-language design approach has been employed. The result is a high-level scripting language with support for both arbitrary precision mathematics, and sieve device control.

## 6.2 Results and Conclusions

Using CASSIE, 12 previously unknown pseudosquare results were obtained, offering further numerical evidence that existing unconditional primality testing algorithms may be improved to $O((\log N)^{3+o(1)})$.

A two-processor (single workstation) software implementation of CASSIE was able to achieve canvass rates of $2.06 \times 10^{15}$ trials per second on the pseudosquare problem putting it ahead of even the fastest dedicated hardware sieve (currently, the Star

Bridge Hypercomputer, which is predicted to achieve a canvass rate of $39 \times 10^{12}$ trials per second for the same problem.)

When combined with 180-processors of the University of Calgary's ACL, a canvass rate of $1.05 \times 10^{18}$ trials per second was obtained for the pseudosquare problem—a result that was at least an order of magnitude above expectations for a software sieve, even with 180-fold parallelism.

With these new pseudosquare results, the modified Selfridge-Weinberger primality test described in Theorem 5.4, combined with trial division up to $2^{20}$ offers one of the fastest known methods for proving the primality of integers up to $2^{100}$. Two applications involving this test were described: a Rabin-Williams signature scheme with extremely fast verifications, and a computational solution to the unsolicited commercial email (spam) problem.

Finally, CASSIE was able to obtain a 97-digit prime, $p$, for which the minus (relative) class number of an associated imaginary cyclic quartic field exceeded 1, answering a challenge originally posed by Louboutin.

## 6.3 Future Improvements

### 6.3.1 Hardware Improvements

The reprogrammable hardware (FPGA) component of CASSIE has not yet been brought online. When it is, it is expected that the sieve rates of each of the doubly-focused sieve problems may be improved. It is not clear yet, however, if the simultaneous enumeration algorithm will act as a bottleneck, limiting any gains the hardware may offer.

A significant improvement may be realized by building the simultaneous enumeration algorithm into hardware, via a pair of pipelined multipliers and a subtractor. Unfortunately, the single Xilinx of the Nallatech Ballynuey board is not large enough to accommodate a design of this size. Further investigations may reveal if adding additional Xilinx modules to the Nallatech board[1] may make this approach realizable. Certainly, a more powerful hardware architecture, such as the Star Bridge Hypercomputer design employed by Wake, *et al.*[WB03] may be flexible enough to implement this idea; dedicating two Processing Elements to the individual sieve problems, and the remaining elements to the multiplier circuitry.

### 6.3.2 Software Improvements

From Bernstein's description [Ber04] of his own software sieve implementation, it seems that at least a 2-fold improvement in sieving speeds may be achievable through further optimization of the software sieve code.

If the Tcl-based command language is not to taste, the use of SWIG to generate the bindings between the CASSIE library and the scripting language make it possible to target additional languages. It would be relatively straightforward to produce Python, Perl, or even PHP-flavoured variants of the CASSIE command language.

[1]The Nallatech Ballynuey 3 board employed by CASSIE can accommodate up to 4 Xilinx devices

# Bibliography

[AB] Swox AB. GNU MP library. http://www.swox.com/gmp/.

[AGS] Hal Abelson, Philip Greenspun, and Lydia Sandon. Tcl for web nerds. http://philip.greenspun.com/tcl/.

[AKS02] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. (preprint), August 2002.

[APR83] Leonard M. Adleman, Carl Pomerance, and Robert S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics. Second Series*, 117(1):173–206, 1983.

[Atk65] A. O. L. Atkin. On pseudo-squares. *Proceedings of the London Mathematical Society (3)*, 14a:22–24, 1965.

[Bac90] Eric Bach. Explicit bounds for primality testing and related problems. *Mathematics of Computation*, 55(191):355–380, 1990.

[Bac97] Adam Back. Hashcash—a denial of service counter-measure. http://www.cypherspace.org/#adam/hashcash/, 1997.

[BB94] Nathan D. Bronson and Duncan A. Buell. Congruential sieves on FPGA computers. In Walter Gautschi, editor, *Mathematics of computation, 1943–1993: a half-century of computational mathematics: Mathematics of Computation 50th Anniversary Symposium, August 9–13, 1993, Vancouver, British Columbia*, volume 48, pages 547–551, Providence, RI, USA, 1994. American Mathematical Society.

[Bea96] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, Monterey, California, USA, July 10–13 1996. USENIX Association.

[Bee39] N.G.W.H. Beeger. Report on some calculations of prime numbers. *Nieuw Archief voor Wiskunde*, 20(2):48–50, 1939.

[Bee46] N.G.W.H Beeger. Note sur la factorisation de quelques grands nombres. *Institut Grand-Ducal de Luxembourg, Sec. d. Sc. nat. phys. math. Archives*, 16:93–95, 1946.

[Ber01] D. J. Bernstein. Enumerating solutions to $p(a) + q(b) = r(c) + s(d)$. *Mathematics of Computation*, 70:389–394, 2001.

[Ber02] D. J. Bernstein. Proving primality after Agrawal-Kayal-Saxena. `http://cr.yp.to/papers/aks.pdf`, 2002.

[Ber03a] D. J. Bernstein. More news from the Rabin-Williams front. `http://cr.yp.to/talks/2003front2.ps`, November 2003. Conference Talk, Mathematics of Public Key Cryptography (MPKC) 2003. University of Illinois at Chicago.

[Ber03b] D. J. Bernstein. Proving tight security for standard Rabin-Williams signatures. `http://cr.yp.to/papers.html#rwtight`, September 2003.

[Ber04] D. J. Bernstein. Doubly focused enumeration of locally square polynomial values. *High Primes and Misdemeanors—Conference in Number Theory in honour of Professor Hugh Williams*, 2004. To appear.

[BH93] Eric Bach and Lorenz Huelsbergen. Statistical evidence for small generating sets. *Mathematics of Computation*, 61(203):62–89, 1993.

[BLS+02] John Brillhart, D.H. Lehmer, J.L. Selfridge, Bryant Tuckerman, and S.S. Wagstaff Jr. *Factorizations of $b^n \pm 1$, b=2,3,5,6,7,10,11,12 Up to High Powers*, volume 22 of *Contemporary Mathematics*. American Mathematical Society, 2002.

[BR98] R. Balasubramanian and D. S. Ramana. Atkin's theorem on pseudo-squares. *Institut Mathématique. Publications. Nouvelle Série*, 63(77):21–25, 1998.

[Bri] John Brillhart. Private correspondence.

[Bri92] John Brillhart. Derrick Henry Lehmer. *Acta Arithmetica*, 62(3):207–220, 1992.

[BS67] John Brillhart and J. L. Selfridge. Some factorizations of $2^n \pm 1$ and related results. *Mathematics of Computation*, 21(97):87–96, 1967.

[BS93] Eric Bach and Jonathan Sorenson. Sieve algorithms for perfect power testing. *Algorithmica*, 9:313–328, 1993.

[BS03] D. J. Bernstein and Jonathan P. Sorenson. Modular exponentiation via the explicit Chinese Remainder Theorem. `http://cr.yp.to/papers.`

`html#meecrt`, September 2003.

[Car20a] E.-O. Carissan. Machine à résoudre les congruences. *Bulletin de la société d'Encouragement pour l'Industrie Nationale*, 132:600–607, 1920.

[Car20b] E.-O. Carissan. Principe méchanique et description de la machine du Ct. E. Carissan. Unpublished, 9 May 1920.

[car33] Machine performs difficult mathematical calculations. *News Service Bulletin*, III(3):19–22, 1933. Carnegie Institution of Washington. Washington, D.C.

[CEFT62] D. G. Cantor, G. Estrin, A. S. Fraenkel, and R. Turn. A very high-speed digital number sieve. *Mathematics of Computation*, 16(78):141–154, 1962.

[Cli48] R. F. Clippinger. A logical coding system applied to the ENIAC (Electronic Numerical Integrator and Computer). Technical Report 673, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland, 29 September 1948.

[Cob66] Alan Cobham. The recognition problem for the set of perfect squares. Technical report, IBM Research, April 1966.

[Col03] F. N. Cole. On the factoring of large numbers. *Bulletin of the American Mathematical Society*, pages 134–137, December 1903.

[dF94] Pierre de Fermat. *Oeuvres de Fermat*, volume 2. Gauthier-Villars et fils, 1894.

[DN92] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. *Advances in Cryptology CRYPTO '92*, (LNCS 740):139–147, 1992.

[fol] FOLDOC: The free online dictionary of computing. `http://www.foldoc.org/foldoc/`.

[FS97] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modelling Language.* Addison Wesley Longman, Inc., 1997.

[GÍ2] A. Gérardin. Question 335. *Sphinx-Oedipe*, 7:47–48, 1912.

[G3́7] A. Gérardin. Machine à congruences (modèle 1937). 70$^e$ *Congrès des Société Savantes de Paris et des Départements*, pages 14, II pp. 37, 1937.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, Massachusetts, USA, 1994.

[Hal33] Marshall Hall. Quadratic residues in factorization. *Bulletin of the American Mathematical Society*, 39:758–763, 1933.

[Hur] Alex Hurwitz. Private correspondence.

[Hus97] H.D. Huskey. SWAC—Standards Western Automatic Computer: the Pioneer Day session at NCC. July 1978. *Annals of the History of Computing, IEEE*, 19(2):51–61, 1997.

[HW79] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, fifth edition, 1979.

[IT89] R. N. Ibbett and N. P. Topham. *Architecture of High Performance Computers*, volume II: Array processors and multiprocessor systems. Springer-Verlag, 1989.

[JB99] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. pages 151–165, 1999.

[JWW04] M.J. Jacobson Jr., H.C. Williams, and Kjell Wooding. Minus class numbers of imaginary cyclic quartic fields. In *Proceedings of the Sixth International Symposium, ANTS-VI*, Lecture Notes in Computer Science. Springer, 2004. (Submitted for publication).

[Kna99] Anthony W. Knapp. Frank Nelson Cole. *Notices of the American Mathematical Society*, 46(8):860, September 1999.

[Kra24] Maurice Kraitchik. *Récherches sur la Théorie des Nombres. Tome I.* Gauthier-Villars, Paris, 1924.

[Kra29] Maurice Kraitchik. *Récherches sur la Théorie des Nombres. Tome II.* Gauthier-Villars, Paris, 1929.

[Law96] F. W. Lawrence. Factorisation of numbers. *Quarterly Journal Pure and Applied Mathematics*, 28:285–311, 1896.

[Leh] D. H. Lehmer. A half-hour discussion of the delay line sieve from the viewpoint of the operator-mathematician. Typewritten transcript.

[Leh28] D. H. Lehmer. The mechanical combination of linear forms. *American Mathematical Monthly*, 35(4):114–121, 1928.

[Leh30] D. H. Lehmer. A fallacious principle in the theory of numbers. *Bulletin of the American Mathematical Society*, 36:847–850, 1930.

[Leh34] D. H. Lehmer. A machine for combining sets of linear congruences. *Mathematische Annalen*, 109(5):661–667, 1934.

[Leh36] D. H. Lehmer. On the converse of Fermat's Theorem. *The American Mathematical Monthly*, 43(6):347–354, 1936.

[Leh46] D. H. Lehmer. Preliminary proposal for the design and construction of the electronic sieve. U.C. Berkeley, unpublished, December 1946.

[Leh49] D. H. Lehmer. On the converse of Fermat's Theorem II. *The American Mathematical Monthly*, 56(5):300–309, 1949.

[Leh53] D. H. Lehmer. The sieve problem for all-purpose computers. *Mathematical Tables and Other Aids to Computation*, 7(41):6–14, 1953.

[Leh54] D. H. Lehmer. A sieve problem on pseudo-squares. *Mathematical Tables and Other Aids to Computation*, 8(48):241–242, 1954.

[Leh58] Emma Lehmer. Criteria for cubic and quartic residuacity. *Mathematika*, 5:20–29, 1958.

[Leh66] D. H. Lehmer. An announcement concerning the delay line sieve DLS 127. *Mathematics of Computation*, 20(96):645–646, October 1966.

[Leh76] D. H. Lehmer. Exploitation of parallelism in number theoretic and combinatorial calculation. In B. L. Hartnell and H. C. Williams, editors, *Proceedings of the Sixth Manitoba Conference on Numerical Mathematics*, pages 95–111. Utilitas Mathematica, 1976.

[Leh80] D. H. Lehmer. A history of the sieve process. In N. Metropolis, J. Howlett, and Gian-Carlo Rota, editors, *A History of Computing in the Twentieth Century*, pages 445–456. Academic Press, Inc., 1980.

[LLS70] D. H. Lehmer, Emma Lehmer, and Daniel Shanks. Integer sequence having prescribed quadratic character. *Mathematics of Computation*, 24(110):433–451, 1970.

[Lou98] S. Louboutin. Computation of relative class numbers of imaginary abelian number fields. *Experimental Mathematics*, 7:293–303, 1998.

[LPW95] R. F. Lukes, C. D. Patterson, and H. C. Williams. Numerical sieving devices: Their history and some applications. *Nieuw Archief voor Wiskunde. Vierde Serie*, 13:113–139, 1995.

[LPW96] Richard F. Lukes, Cameron D. Patterson, and H. C. Williams. Some results on pseudosquares. *Mathematics of Computation*, 65(213):361–372, S25–S27, 1996.

[Luk95] Richard F. Lukes. *A Very Fast Electronic Number Sieve.* PhD thesis, The University of Manitoba, 1995.

[MB75] Michael A. Morrison and John Brillhart. A method of factoring and the factorization of $f_7$. *Mathematics of Computation*, 29(129):183–205, 1975.

[MvOV96] A. Menezes, P van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 5 edition, 1996.

[Nol] Landon Curt Noll. C-style arbitrary precision calculator. `http://www.isthe.com/chongo/tech/comp/calc/`.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[Ous98] John K. Ousterhout. Scripting: higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[Pat83] Cameron Douglas Patterson. Design and use of an electronic sieve. Master's thesis, University of Manitoba, 1983.

[Pat92] Cameron Douglas Patterson. *Derivation of a High Speed Sieve Device.* PhD thesis, The University of Calgary, 1992.

[Pom82] Carl Pomerance. The search for prime numbers. *Scientific American*, 247(6):136–147, December 1982.

[RSW96] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS/TR-684, 1996.

[Sch97] A. Schinzel. On pseudosquares. *New Trends in Probility and Statistics*, 4:213–220, 1997.

[See86] P. Seelhoff. Ein neues kennzeichen für die primzahlen. *Zeitschrift für Mathematik und Physik*, 31:306–310, 1886.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[Sta95] William Stallings. How to certify public keys without a central authority. *Byte Magazine*, 1995.

[Ste89] Allan J. Stephens. *OASiS: An Open Architecture Sieve System for Problems in Number Theory.* PhD thesis, The University of Manitoba, 1989.

[SW90] A. J Stephens and H. C. Williams. An open architecture number sieve. In *Number Theory and Cryptography (Sydney, 1989)*, volume 154 of *London Math. Soc. Lecture Note Ser.*, pages 38–75. Cambridge Univ. Press, Cambridge, 1990.

[SWM95] J. Shallit, H. C. Williams, and F. Morain. Discovery of a lost factoring machine. *The Mathematical Intelligencer*, 17(3):41–47, Summer 1995.

[tRW03] H.J. te Riele and H.C. Williams. New computations concerning the Cohen-Lenstra heuristics. *Experimental Mathematics*, 12(1):99–113, 2003.

[TW99] E. Teske and H.C. Williams. A problem concerning a character sum. *Experimental Mathematics*, 8(1):63–72, 1999.

[WB03] H. A. Wake and D. A. Buell. Congruential sieves on a reconfigurable computer. *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 11–18, April 2003.

[Wed01] Sebastian Wedeniwski. *Primality Tests on Commutator Curves.* PhD thesis, Eberhard-Karls-Universität, Tübingen, 2001.

[Wel97] Brent B. Welch, editor. *Practical Programming in Tcl and Tk.* Prentice Hall, 2 edition, 1997.

[Wil78] H. C. Williams. Primality testing on a computer. *Ars Combinatoria*, 5:127–185, 1978.

[Wil80] H. C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, 1980.

[Wil98] H. C. Williams. *Édouard Lucas and Primality Testing*, volume 22 of *Canadian Mathematical Society Series of Monographs and Advanced Texts.* Wiley-Interscience, 1998.

[WM68] A. E. Western and J. C. P. Miller. *Tables of Indices and Primitive Roots*, volume 9 of *Royal Society Mathematical Tables.* Published for the Royal Society at the Cambridge University Press, London, 1968.

# Appendix A

# CASSIE User's Manual

## A.1 Introduction

The CASSIE control language is a superset of Tcl, the Tool Command Language. A brief introduction to Tcl will now be given. For a more detailed treatment, see [Wel97] or [AGS].

### A.1.1 Tcl Overview

As a programming language, Tcl is quite Lisp-like (a resemblance for which I apologize in advance). Tcl syntax is extremely simple, consisting of a procedure name followed by its arguments. *i.e.*

```
procedure_name arg1 arg2 arg3
procedure_name arg1 [subprocedure subarg1 subarg2]
```

Sub-procedures invocations are denoted by enclosing them in square brackets, as in `[subprocedure subarg1 subarg2]`. Variables names are indicated by prefixing them with a dollar sign, *i.e.*`$variable`.

In Tcl, white space is used as an argument separator. If white space is intended to be part of a string literal, grouping delimiters must be used. There are two sets of grouping delimiters in Tcl: double quotes (`""`) and curly braces ({}). These delimiters differ in that variable and subcommand interpolation is performed in quoted strings, but not in text enclosed by curly braces. *i.e.*

```
set i 1
puts "$i"
puts {$i}
puts "[expr 2 + 2]"
puts {[expr 2 + 2]}
```

will produce the output:

```
1
$i
4
[expr 2 + 2]
```

Here, the puts function simply prints a string to the standard output device. The expr evaluates an arithmetic expression in infix notation.

There are two main data types of interest in Tcl: strings and lists. In Tcl, every data type has a string representation, and thus effectively, Tcl treats every piece of data as a string unless instructed otherwise.

Lists in Tcl may be formed by enclosing space-delimited elements in curly braces. For instance, the string

```
{7 {1 2 4}}
```

may be treated as a list containing two elements: "7" and "{1 2 4}". Of course, the string {1 2 4} may be subsequently treated as a list containing the three elements "1", "2", and "4". This nesting of lists allows for the creation of more complex data types.

Lists are decomposed into their constituent strings using the `lindex` procedure. List elements are indexed starting at zero. For example, the commands

```
set l {this is a {list example}}
puts [lindex $l 0]
```

will display the string "this", while

```
set l {this is a {list example}}
puts [lindex $l 3]
```

displays the string "list example".

## A.1.2 Using Objects

Though object-oriented flavours of Tcl exist,[1] Tcl itself—and thus the CASSIE command language—is is a strictly procedural language. One common technique for achieving a semblance of object-orientation is to use a command-subcommand mechanism for procedures. *i.e.*

```
object method arg1 arg2
```

This technique is used throughout the CASSIE command language. Complex data structures implemented and manipulated in this manner will be referred to as *objects* throughout this manual.

In general, objects are instantiated using a command of the form:

```
object_type <object_name>
```

*i.e.*to create a ring named ring0

```
ring ring0
```

Attributes for a given object may be set using the *configure* command, followed by a flag representing the attribute name and its desired value. Multiple attributes may be set with a single *configure* command, as in the following:

[1]See, for example, the [incr tcl] project at http://www.tcltk.com/itcl

```
<object_name> configure -paramater1 value1 -parameter2 value2 ...
```

*i.e.*to create a `monitor` object, and set its *logfile* and *rptfile* attributes to appropriate values:

```
monitor m
m configure -logfile cassie.log -rptfile cassie.rpt
```

Attributes may be examined by invoking the *cget* method with the attribute name as a flag. For example, to see the current *interval* setting for the previously defined `m` object:

```
m cget -interval
```

Methods may be invoked by simply specifying the method name (and any parameters) after the object name. *i.e.*to create a `sieve` called `pseudosquare`, attach two rings, and run the problem, the following script could be used:

```
sieve pseudosquare
pseudosquare ring_add {{8 1} {3 1}}
pseudosquare ring_add {{5 {1 4}} {7 {1 2 4}}}
pseudosquare run
```

In many cases, object-specific methods will be introduced to modify the underlying object attributes, as directly modifying certain objects may result in a sieve representation that is internally inconsistent. For this reason, use of the *configure* method for setting sieve attributes should be avoided, where possible.

For example, while it is certainly possible to set the *modulus* and *residue* attributes of a `sieve` object using a sequence of commands like:

```
sieve s
s configure -modulus 24 -residue 1
```

It is much safer to use the *normalize* method of the `sieve` object to perform the same task, as any attached sieve rings will be affected by the change:

```
sieve s
s normalize 24 1
```

### A.1.3 CASSIE Object Hierarchy

The main user-modifiable object used by CASSIE is the `sieve` object. Most user interaction is handled by this object. Under the hood, a number of additional objects are used, including

- `monitor` – Object containing information pertinent to logging, checkpointing, and monitoring a particular sieve run.
- `ring` – Object containing a modulus and its associated (acceptable) residues.
- `ssieve` – A singly-focused sieve implementation.
- `scoreboard` – A list of moduli and their associated "best" sieve values.

The relationships between these objects can be shown using a Unified Modeling Language (UML) diagram[FS97], as is done in Figure A.1.

The various CASSIE control language objects will now be described in more detail.

## A.2 Sieve Object Detail

Most of the user's interaction with the sieve will occur via the `sieve` object.

The `sieve` object is the main container for a sieve problem. It encapsulates the sieve implementation ( one or two `ssieve` objects), checkpoints (`monitor` object), results (`scoreboard` object), filter procedures (the *filter_proc* attribute), and

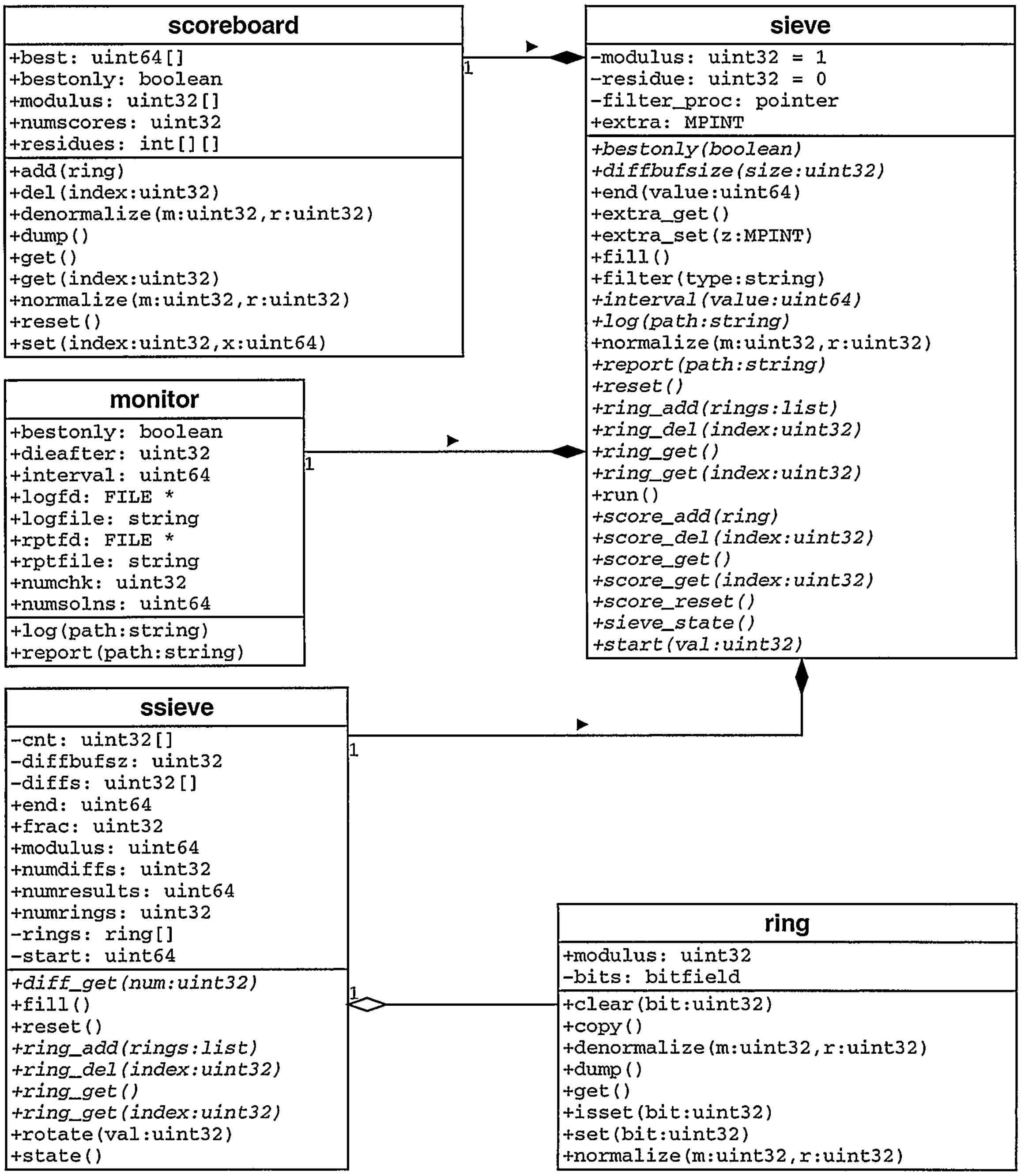

*Figure A.1*: UML Class Diagram for Sieve Objects

optimization parameters (*modulus*, *residue* attributes) into a single object. From a design pattern standpoint[GHJV95], the sieve object is a facade for the underlying sieve implementation.

### A.2.1 Attributes

The following attributes comprise the `sieve` object:

**modulus** : *uint32* – Modulus used for Lehmer normalization[Leh53]. (Default: 1). Do not set this value manually. Instead, use the *normalize/denormalize* methods, described below.

**residue** : *uint32* – Residue used for Lehmer normalization. (Default: 0). This value should not exceed (or equal) the value of the *modulus* attribute. Do not set this value manually. Instead, use the *normalize/denormalize* methods described below.

**filter_proc** : *function pointer* – Pointer to a C function that will filter the data emerging from the sieve. Do not modify this field directly. Use the *filter* method described below.

**extra** : *multiprecision* – This is a multipurpose field used to pass an optional filter-specific parameter to sieve filter procedures. *i.e.*The `abprime` filter uses this parameter to pass the value of $a^2$ so that the value $a^2 + b^2$ ($b$ is obtained from the sieve) can be tested for primality.

### A.2.2 Methods

The following methods are defined for the `sieve` object. Note that many of these methods simply act as proxies for methods of the underlying `ssieve`, `monitor`, `ring`, and `scoreboard` objects:

**bestonly : *boolean*** – (Default 0) If set to 0, all acceptable solutions found by the sieve will be written to the associated output file. If 1, only values better than the *best* entries of the associated `scoreboard` object are written. This functionality is useful primarily in doubly-focused sieves, where sieve solutions emerge from the sieve in essentially random order.

**diffbufsize *size* : *uint32*** – The number of sieve outputs (stored as integer differences) to obtain each time the sieve buffer is filled. Because sieve solutions are obtained by examining solution taps in parallel, there is a lower bound (currently 33) on this parameter. For small, sparse sieve problems, smaller diff buffers may be desirable to avoid sieving much beyond the stated endpoint of the problem.

**end *value* : *uint32*** – Modify the upper bound value for the sieve problem. (Stored internally as the *end* attribute of the `ssieve` object). This method returns the old *end* value.

**extra_get** – Return the current value of the *extra* attribute.

**extra_set *value* : *multiprecision*** – Set the value of the *extra* attribute to the supplied multiprecision integer value.

**fill** – Fill up to *diffbufsz* values into the underlying `ssieve` objects *diff* buffer. Values are entered into the buffer as differences from the previous sieve output. The sieve is operated from its current state, and the *numdiffs* attribute of the underlying `ssieve` object is updated to reflect the number of values that were placed into the *diff* buffer. This method is normally used only for debugging. For normal operation, see the *run* method.

**filter *filter_type*** – Add a filter to the sieve outputs.

Filters are C functions with a declaration matching the prototype:

```
int filter_proc(unsigned long long x, struct sieve *sv);
```

Filters are intended to return the value FILTER_YES if the filter condition is matched (and hence, the indicated $x$ value should be *excluded* from consideration) and FILTER_NO otherwise. These return values are defined in the `filters.h` header file.

Filters are defined in the `filters.c` file, and its associated header file. Currently supported filter types are:

**none** – do not filter.

**abprime** – accept only (probable) prime values of the form $a^2 + b^2$, where $b$ is the value obtained from the sieve, and $a^2$ is passed via the *extra* attribute.

**perfect_square** – exclude perfect squares.

**interval *interval* : *uint32*** – The number of sieve fills between checkpoints (default: 10000). This parameter is typically adjusted to produce a checkpoint every hour.

**log** ***filename*** **:** ***string*** – Set log file name, or "none."

**normalize** ***modulus*** **:** ***uint32*****,** ***residue*** **:** ***uint32*** – Normalize the sieve problem using Lehmer's single residue optimization technique[Leh53]. All attached rings and scoreboard moduli are normalized, and the *modulus* and *residue* attributes of the `sieve` object are updated. If a normalization is already in effect this normalization is first removed, and the new normalization applied. Thus, applying a normalization using *modulus* 1 and *residue* 0 effectively removes (denormalizes) any optimizations that may have been applied.

**report** ***filename*** **:** ***string*** – Set report file name, or "none."

**reset** – Reset sieve counters (and internal state variables) to their appropriate start values.

**ring_add** ***rings*** **:** ***list*** – Adds a ring to the underlying `ssieve` object. Rings may be specified using the Tcl list notation, either:

```
{modulus {residue1 residue2 residue3 ...}}
```

or

```
{m_1 {r_11 r_12 r_13 ...} ... {m_j {r_1j r_2j r_3j ...}}}
```

If multiple rings are specified using the second form, they will be automatically combined (using the Chinese Remainder Theorem) into a single modulus and list of acceptable residues. For example:

```
sieve0 ring_add {{8 1} {3 1} {5 {1 4}}}
```

will actually add the ring:

```
{120 {1 49}}
```

**ring_del** *index* : *uint32* – Delete the indicated ring from the underlying `ssieve` object. Rings indexes are specified starting at 1.

**ring_get** *index* : *uint32* – Return the ring corresponding to *index*. If no ring index is supplied, all rings will be returned in the form of a Tcl list-of-lists.

**run** – Run the sieve problem, based on the parameters defined in this `sieve` object.

**score_add** *ring* – Add the specified `ring` to the scoreboard, and set its associated best value to the default (currently $2^{64} - 1$).

**score_del** *index* : *uint32* – Remove the indicated scoreboard entry from the associated `scoreboard` object. Scoreboard entries are indexed starting at 1.

**score_get** *index* : *uint32* – Return the indicated scoreboard entry. If no scoreboard index is supplied, the entire scoreboard will be returned, in the form of a Tcl list-of-lists.

**score_reset** – Reset all *best* values for the associated `scoreboard` object to BEST_DEFAULT, currently $2^{64} - 1$.

**sieve_state** – Return the internal state of the sieve. Currently, this is a Tcl list of the form:

```
{{ring counters} frac numresults}
```

This method is intended for debugging only.

**start** ***val*** **:** ***uint64*** – Rotate the sieve rings to a new start position, and update the *ssieve.start* attribute to reflect this new value. Any existing rotation is removed before the new start value is applied.

### A.2.3 Examples

To create a new sieve instance named *spoon*:

```
sieve spoon
```

To add several rings (moduli and their associated acceptable residues) to this sieve:

```
spoon ring_add {5 {1 4}}
spoon ring_add {7 {1 2 4}}
```

To set the normalization modulus and residue for this sieve instance:

```
spoon normalize  24 1
```

To add a perfect square filter:

```
spoon filter perfect_square
```

To enable logging to a file called `spoon.out`:

```
spoon log spoon.out
```

To fill up to *diffbufsz* (default 1024) values into the *diff* buffer:

```
spoon fill
```

To reset the sieve, and start sieving at the value 10,000:

```
spoon reset
spoon start 10000
```

Finally, to run this sieve instance:

```
spoon run
```

## A.3 Underlying Object Detail

Most user interaction with sieve objects should occur via the attributes and methods of the `sieve` object. For certain advanced applications, however, it may be necessary to interact with the underlying implementation. This section describes the objects that comprise that implementation.

### A.3.1 Ring Object

The ring object corresponds to the sieve ring: the list of acceptable residues for a particular modulus, or set of moduli. Rings are not usually manipulated directly. Instead, the *ring_add* and *ring_del* methods of the parent `sieve` or `ssieve` objects are typically used.

#### Attributes

**modulus** : *uint32* – The modulus (or product of moduli) represented by this ring. This attribute should not be modified directly. See the magic constructor below.

**bits** : *bitfield* – This is the structure used to represent the acceptable residues for a particular modulus. It should not be modified directly. See the *clear*, *set*, *get* and *dump* methods below.

#### Methods

**The magic constructor** –

The ring object uses a special constructor to accept ring values in the form of a Tcl list defining the modulus and acceptable residues. The Tcl list is of the form:

```
{modulus {residue1 residue2 residue3 $\ldots$}}
```

or

```
{m_1 {r_11 r_12 r_13 ...} ... {m_j {r_1j r_2j r_3j ...}}}
```

The magical part of this constructor is the following: if a list of modulus/residue pairs is given to the constructor, they will be automatically merged into a single ring consisting of the intersection of all acceptable moduli. In other words, the moduli obtained by the Chinese Remainder combination of the listed acceptable residues/modulus pairs, as described in section 1.2.1 *i.e.*

```
{ {8 1}
  {3 1}
  {5 {1 4}}
  {7 { 1 2 4}}
}
```

Will be reduced to the equivalent:

```
840 { 1 121 169 289 361 529 }
```

**dump** – Dump the internal representation (currently a bitfield) of this ring. This method is intended for testing purposes only. In practice, an equivalent, and more compact result can be obtained using the *get* method.

**clear** ***index* : *uint32*** – Remove *index* from the acceptable residue list. In other words, clear bit number *index* in the internal bitfield representation.

form:

```
{modulus {residue1 residue2 residue3 $\ldots$}}
```

or

```
{m_1 {r_11 r_12 r_13 ...} ... {m_j {r_1j r_2j r_3j ...}}}
```

The magical part of this constructor is the following: if a list of modulus/residue pairs is given to the constructor, they will be automatically merged into a single ring consisting of the intersection of all acceptable moduli. In other words, the moduli obtained by the Chinese Remainder combination of the listed acceptable residues/modulus pairs, as described in section 1.2.1 *i.e.*

```
{ {8 1}
  {3 1}
  {5 {1 4}}
  {7 { 1 2 4}}
}
```

Will be reduced to the equivalent:

```
840 { 1 121 169 289 361 529 }
```

**dump** – Dump the internal representation (currently a bitfield) of this ring. This method is intended for testing purposes only. In practice, an equivalent, and more compact result can be obtained using the *get* method.

**clear** ***index* : *uint32*** – Remove *index* from the acceptable residue list. In other words, clear bit number *index* in the internal bitfield representation.

**set** ***index*** **:** ***uint32*** – Add *index* to the acceptable residue list. In other words, set bit number *index* in the internal bitfield representation to 1.

**copy** – Return a copy of this ring object. Tcl is not well suited to passing pointers around, so making a copy is the safest way to make use of a ring structure programmatically.

**get** Return a string representation (in the form of a Tcl list) of the ring contents, in the combined form:

```
{modulus {residue1 residue2 ... residuek}}
```

**Examples**

To create a ring called `ring2` made up of the moduli 5, 7, 11, 13, 17 (and their acceptable residues):

```
ring ring2 {
  {5 {1 4}}
  {7 {1 2 4}}
  {11 {1 3 4 5 9}}
  {13 {1 3 4 9 10 12}}
  {17 {1 2 4 8 9 13 15 16}}
}
```

To retrieve the combined representation of this ring:

```
ring2 get
```

In this case, the output (having been combined into a single ring via the Chinese Remainder Theorem) will look something like the following:

```
85085 { 1 4 9 16 36 64 81 144 179 191 246 256 324 361 389 529 576
 599 716 729 764 841 914 939 961 984 1024 1101 1114 1171 1226 1296
 ...(many lines omitted)...
 84254 84319 84379 84386 84639 84659 84681 84709 84764 84984 84991 }
```

The CRT combination of inputs makes the ring object useful when computing normalization moduli and residues. For example, the following code computes the normalization function, $x = my + r$ from a set of four moduli and their associated acceptable residues. The various choices for $r$ are selected by choosing subsequent values for $\$n$:

```
ring nv [list [make_ring 7] \
              [make_ring 11] \
              [make_ring 13] \
              [make_ring 17]]
set normvector [nv get]
set m [lindex $normvector 0]
set r [lindex [lindex $normvector 1] $n]
```

This example assumes the existence of a `make_ring` function, which computes the acceptable residues for a particular modulus, returning the result in the form:

```
     {modulus {residue1 residue2 ... residuek}}
```

For example, the `make_ring` function for the pseudosquare problem could be written as follows:

```
# Given a modulus p, return all residues that are quadratic residues
# modulo p. i.e. e_i = 1 for all p <= p_i
proc make_ring {p} {
    set res_l [list 1] ;# 1 is always a Quadratic Residue

    for {set i 2} {$i < $p} {incr i} {
        if {[JACOBI $i $p] == 1} {
            lappend res_l $i
        }
    }
    return [list $p $res_l]
}
```

### A.3.2 Ssieve Object

The `ssieve` object represents a basic implementation of the General Sieve Problem, including a set of bounds (*start*, *end*), a set of rings (moduli with acceptable residues), a buffer containing sieve outputs (*diffs*, *diffbufsz*), and the internal state needed to operate the sieve. A traditional (singly-focused) sieve implementation makes use of one `ssieve` object. A doubly-focused sieve uses two underlying `ssieves`.

#### Attributes

The following attributes are defined for the `ssieve` object:

**start** : *uint64* – The lower bound of the sieve problem (the $A$ in $A \leq x < B$). This attribute should not be modified directly, as changing the sieve's start position typically requires rotating all attached rings to a new starting position. Use *rotate* method (described below) to safely modify this attribute.

**end** : *uint64* – The upper bound of sieve problem (the $B$ in $A \leq x < B$). It is safe to modify this attribute using the *configure* method described in section A.1.2.

**diffbufsz** : *uint32* – The maximum number of sieve outputs to generate per invocation of *fill.* The default size is 1024 entries.

**numrings** : *uint32* – Number of rings attached to this sieve. Do not modify this attribute directly. It is updated automatically by the *ring_add* and *ring_del* methods.

**internal state** : *various* – The other attributes in the sieve object are used to keep track of internal sieve state and should never be accessed directly. They are therefore omitted from this manual.

## Methods

The following methods are defined for the `ssieve` object:

**fill** – Fill up to *diffbufsz* values into the `ssieve` object's *diff* buffer. Values are entered into the buffer as differences from the previous sieve output. The sieve is operated from its current state, and the *numdiffs* attribute of the underlying `ssieve` object is updated to reflect the number of values that were placed into the *diff* buffer. This method is normally used only for debugging.

**ring_add *ring* : *list*** – Add a ring to the sieve. Rings are always copied before being attached to the sieve, with the copy being attached to the `ssieve` object, as the ring contents will be automatically rotated, based on the value of the *start* attribute. The *ring_add* method employs the magic ring constructor (described in A.3.1), and thus accepts a Tcl string as input.

**ring_del *ring_no* : *uint32*** – Remove a ring from the sieve. This method automatically updates the *numrings* attribute.

**ring_get *ringno* : *uint32*** – Return the indicated ring number. If no ring number is supplied, all rings will be returned in the form of a Tcl list-of-lists.

**diff_get *num*** – returns the first *num* entries in the diff array as a Tcl list.

**reset** – Reset the internal state of the sieve to its starting state. currently, the state is comprised of the *frac* field, and one *cnt* (counter) field for each of the attached rings.

**rotate** – Modify the start position of the sieve. All attached rings are rotated to this start position, and their acceptable residues modified accordingly.

**state** – Return the internal state of the sieve. This method is intended for debugging purposes only.

**start** ***val* : *uint64*** – Adjust the start position of the sieve. All attached rings are rotated to the new start position. (note that the attached rings are actually rotated by the difference between the old and new start values)

### Examples

Build the sieve.

```
ssieve mysieve
mysieve start 0 ;# optional. 0 is default
mysieve configure -end 10000
```

Add two rings (previously defined) rings to the sieve:

```
mysieve ring_add r1
mysieve ring_add r2
```

See all rings attached to the sieve:

```
mysieve ring_get
```

Remove the second ring, added above:

```
mysieve ring_del 2
```

Look at ring 1 in the sieve defined above:

```
[mysieve ring_get 1] get
```

This last example was a bit tricky. An equivalent, but possibly more confusing notation is:

```
set ring1 [mysieve ring_get 1]
$ring1 get
```

Unfortunately, the example above runs into some scoping bugs in Tcl where the `$ring1` object is sometimes destroyed by an over-eager garbage collector. A safer construct is the following:

```
ring ring1 -this [mysieve ring_get 1]
ring1 get
```

The difference is subtle, wherein the ring object is assigned a real name, `ring1`, and not merely an automatic one as in the `$ring1` example.

### A.3.3 Scoreboard Object

The `scoreboard` object contains a list of moduli and their associated "best" sieve values.

#### Attributes

**modulus : *uint32 array*** – Array containing the `scoreboard` moduli.

**best : *uint64 array*** – Array containing the `scoreboard` "best" values.

**numscores : *uint32*** – The number of entries in the *modulus* and *best* arrays. Do not modify this attribute directly. It will be adjusted automatically when the *add* and *del* methods are used.

#### Methods

**add *modulus* : *uint32*, *best* : *uint64*** – Add an entry to the scoreboard array. If the *best* attribute is omitted, the maximum value (currently $2^{64} - 1$) is assumed.

**del** ***index* : *uint32*** – Delete the *index*th entry from the `scoreboard` array.

**get** ***index* : *uint32*** – Return a string (or Tcl list) version of the *index*th entry from the `scoreboard` array.

**reset** – Reset all scoreboard *best* values to BEST_DEFAULT, currently $2^{64} - 1$.

**Examples**
Create a scoreboard, check the defaults:

```
scoreboard sc
```

Add a ring to the scoreboard:

```
sc add {{8 1} {3 1} {5 {1 4}}}
```

Add a ring (assigning the returned ring number to a variable), and view it:

```
set idx [sc add {7 {1 2 4}}]
sc get $idx
```

View all scoreboard rings:

```
sc get
```

Add a ring, and set its associated *best* value to 12345:

```
set scnum [sc add {11 {0 2 4}}]
sc set $scnum 12345]
```

Reset the scoreboard best values to default and view the result:

```
sc reset
sc get
```

### A.3.4 Monitor Object

This object contains information necessary for the logging and checkpointing of sieve operations.

## Checkpoint Overview

A sieve *checkpoint* is a means of verifying and recording the sieve status at a given point in time. Verification entails testing the internal sieve representation against the original problem parameters, *i.e.*verifying that bit patterns match the residue conditions of the problem. For long-running, parallel problems, there is a non-negligible probability that a bit of ram could be inadvertently flipped due to electrical effects, or even cosmic rays. Recording the sieve status allows a sieve job to be restarted in case of a machine outage, software fault, or verification failure.

Checkpoint information is recorded in a report (`.rpt`) file, which may be specified using the *report* attribute. The format of this file is as follows:

```
{start  {scoreboard_list} {sieve_status}}
{rings {ring_list}}
{chk {scoreboard_list} {sieve_status}}
...
{chk {scoreboard_list} {sieve_status}}
{end  {scoreboard_list} {sieve_status}}
```

The first parameter indicates the entry type: *start*, *rings*, *chk*, or *end*. The *start*, *chk*, and *end* differ only by the text label used to identify the entry.

the `ring_list` is specified as a list of rings. *i.e.*

```
{{modulus1 {residue list 1}} {modulus 2 {residue list 2}} ...}
```

Finally, the `sieve_status` list has the following format:

```
 {sieve counters} frac x
```

Where $x$ is the current value under investigation in the sieve, and *frac* represents the carry-out from the last sieve fill operation.

Attributes

**dieafter :** ***uint32*** – This attribute is used mainly for debugging and regression testing. If nonzero, the sieve run will terminate after *dieafter* checkpoint entries have been written. See also the *numchk* attribute.

**interval :** ***uint64*** – The number of values to sieve before a checkpoint is written.

**logfd:** ***file descriptor*** – Unix file descriptor associated with the log file.

**logfile :** ***string*** – String containing the log (results) file name.

**rptfd:** ***file descriptor*** – Unix file descriptor associated with the report (checkpoint) file.

**rptfile :** ***string*** – String containing the report (checkpoint) file name.

**numchk :** ***uint32*** – Number of checkpoints that have been written. This attribute is used mainly in conjunction with the *dieafter* attribute.

**numsolns :** ***uint32*** – The number of solutions found by the sieve. This attribute is used in conjunction with *interval* to determine when a checkpoint should be written.

Methods

**log** ***path*** **:** ***string*** – Set the name of the log file to which sieve outputs will be written. If "none" is specified, logging functions are disabled.

**report** ***path*** **:** ***string*** – Set the name of the report file to which checkpoints will be written. If "none" is specified, checkpointing functions are disabled.

## A.4 Parallelizing Sieve Problems

In Section 3.2.3, a parallelization technique was described where a sieve problem was partitioned into $|\mathcal{R}_i|$ parallel problems by performing normalization on each of $r_{ij} \in \mathcal{R}_i$ for $0 \leq j < |\mathcal{R}_i|$ acceptable residues, and sieving on each of these problems in parallel. $i.e. x = yM_i + r_{ij}$.

The following sieve framework accepts two parameters on the command line: the sieve instance (indicating which of the normalization residues should be used), and the total number of parallelized processes. It assumes the existence of three external functions: `get_normalization_rings`, which returns a normalization modulus and $|\mathcal{R}_i|$ associated residues in the form of a ring list, `primes`, which returns a list of primes between two bounds, and `make_ring` which returns the modulus and acceptable residues (in the form of a ring list) associated with a given prime.

```
if {[llength $argv] == 2} {
    set num [lindex $argv 0]
    set of [lindex $argv 1]
} else {
    set num 1
    set of 1
}
set normvector [get_normalization_rings]

for {set n [expr ($num - 1)]} \
    {$n < [llength [lindex $normvector 1]]} {incr n $of} {

    set m [lindex $normvector 0]
    set r [lindex [lindex $normvector 1] $n]

    sieve s
    s log "sieve-$m-$r.out"
    s report "sieve-$m-$r.rpt"
    foreach p [primes 7 79] {
      s ring_add [make_ring $p]
      }

    foreach p [primes 79 257] {
      s score_add [make_ring $p]
      }
    s normalize $m $r
    s run
}
```

## A.5 Sample Sieve Problems

This sample sieve solves the Lehmer/Lehmer/Shanks problem number I (*i.e.*the pseudosquare problem) for all primes up to 103:

```
#!/usr/bin/tclsh
### Helper Functions
# sieve-lib provides the multiprecision function JACOBI,
source "sieve-lib.tcl"

# Procedure to implement the LLS-I problem condition:
# Given a modulus p, return all residues that are quadratic residues
# modulo p.
# i.e. e_i = 1 for all p <= p_i

proc psquare_ring {p} {
    set res_l [list 1] ;# 1 is always a Quadratic Residue
    for {set i 2} {$i < $p} {incr i} {
        if {[JACOBI $i $p] == 1} {
            lappend res_l $i
        }
    }
    return [list $p $res_l]
}

# Implementation of Lehmer-Lehmer-Shanks Problem I
# i.e. The pseudosquare problem.
sieve lls1
```

```
# We know from [lls70] that the pseudosqtares from L_3 to L_79 lie
# in the interval 0 - 900,000,000. We can use the 24x+1 optimization
# to reduce this effective interval to 0-37,500,000
lls1 start 0
lls1 end [mpexpr (900000000 - 1) / 24]

# Set up reporting / checkpointing
lls1 log lls1.out
lls1 report lls1.rpt

# Exclude perfect squares
lls1 filter perfect_square

# add the rings. First, the even condition:
lls1 ring_add {8 1}

# Add a ring for each of the primes from 3 to 17.
# This will serve as our exclusion sieve
foreach p [primes 3 17] {
        lls1 ring_add [psquare_ring $p]
}

# Maintain a ``scoreboard'' for all primes between 17 and 127
foreach p [primes 17 127] {
    lls1 score_add [psquare_ring $p]
}

lls1 normalize 24 1
lls1 run

# To display the (denormalized) sieve results, uncomment the following line:
# (Raw sieve results are placed in lls1.out)
# score_print [lls1 cget -this]

# Now, if you want to rerun the problem without the perfect square filter
# do a:
# lls1 filter none
# lls1 reset
# lls1 run
```

This sieve produced the following (denormalized) output in 14.363s on an AMD Athlon 2000+. A total of 16,925,139 values were found that matched the sieve criteria. The effective canvas rate of the sieve was 1,670,960,105 trials per second.

```
17: 18001
19: 53881
23: 87481
29: 117049
31: 515761
37: 1083289
41: 3206641
43: 3818929
47: 9257329
53: 22000801
59: 48473881
61: 48473881
67: 175244281
71: 427733329
73: 427733329
79: 898716289
83: 2805544681
89: 2805544681
97: 2805544681
101: 10310263441
103: 23616331489

real    0m14.363s
user    0m14.240s
sys     0m0.000s
```

The following sieve script shows how a sieve problem may be parallelized, by first creating a sieve ring containing the normalization residues, and then iterating through them.

```
#!/bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
```

```
# Helper Functions
source "sieve-lib.tcl"
source quadmu-lib.tcl

### SIEVE CODE STARTS HERE
# Make the sieve problem parallelizable
if {[llength $argv] == 2} {
    set num [lindex $argv 0]
    set of [lindex $argv 1]
} else {
    set num 1
    set of 1
}

# The following ring contains 4 acceptable residues
ring normring [list [quadmu_ring 7] \
                    [quadmu_ring 11]]

set nv [normring get]

for {set n [expr ($num - 1)]} {$n < [llength [lindex $nv 1]]} {incr n $of} {

    set m [lindex $nv 0]
    set r [lindex [lindex $nv 1] $n]

    sieve qu

    qu start 0
    qu end 1440000000

    # Fairly sparse problem, so use a small sieve buffer
    qu diffbufsize 33

    # Set up reporting / checkpointing
    qu log "qu-$m-$r.out"
    qu report "qu-$m-$r.rpt"

    foreach p [primes 13 41] {
        qu ring_add [quadmu_ring $p]
    }

    # Keep a scoreboard for all primes between 41 and 127
    foreach p [primes 41 127] {
        qu score_add [quadmu_ring $p]
    }
    qu normalize $m $r

    qu run
}
```

If used in conjunction with a grid computing manager, such as the Sun Grid Engine, parallelization of this script may be accomplished by using a shell wrapper (`myjob.sh`) resembling the following:

```
#!/bin/tcsh
sieve.tcl ${SGE_TASK_ID} $1
```

And the sieve may be executed over 4 units in the following manner:

```
qsub -t 1-4 myjob.sh 4
```

Executing the job in this manner produces 4 output files, one for each of the normalization functions:

```
qu-77-13.out
qu-77-20.out
qu-77-57.out
qu-77-64.out
```