מכון ויצמן למדע
WEIZMANN INSTITUTE OF SCIENCE

Thesis for the degree
Doctor of Philosophy

חבור לשם קבלת התואר
דוקטור לפילוסופיה

by

Eran Tromer

מאת

ערן טרומר

Hardware-Based Cryptanalysis

שבירת צפנים באמצעי חומרה

Advisor

Prof. Adi Shamir

מנחה

פרופ׳ עדי שמיר

May 2007

אייר תשס״ז

Presented to the Scientific Council of the
Weizmann Institute of Science
Rehovot, Israel

מוגש למועצה המדעית של
מכון ויצמן למדע
רחובות, ישראל

Summary

The theoretical view of cryptography usually models all parties, legitimate ones as well as attackers, as idealized computational devices with designated interfaces, and their security and computational complexity are evaluated in some convenient computational model – usually PC-like RAM machines. This dissertation investigates several cases where reality significantly deviates from this model, leading to previously unforeseen cryptanalytic attacks.

The first part of the dissertation investigates the concrete cost of factoring integers, and in particular RSA keys of commonly used sizes such as 1024 bits. Until recently, this task was considered infeasible (i.e., its cost was estimated as trillions of dollars), based on extrapolations that assumed implementation of factoring algorithms on sequential PC-like computers. We have shown that the situation changes significantly when one introduces custom-built hardware architectures, with algorithms and parametrization that are optimized for concrete technological tradeoffs and do not fit the RAM machine model. Focusing on the Number Field Sieve (NFS) factoring algorithm, we propose hardware architectures for both of its computational steps: the sieving step and the linear algebra step. Detailed analysis and a careful choice of the NFS parameters show that for breaking 1024-bit RSA keys, NFS can be be implemented at a fairly practical cost of a few million US dollars for a throughput of one factorization per year. This casts grave doubt on the security of such cryptographic keys, which are widely deployed in accordance with existing standards and recommendations.

The second part of the dissertation investigates another abstraction violation: side-channel information leakage from cryptographic systems. We demonstrate two such channels. First, cache contention in modern CPUs leads to leakage of information about memory access patterns, and we devise novel ways to exploit this leakage. The new attacks work in pure software, and are applicable even in scenarios where the attacker has no nominal communication channel with the attacked code. They are also extremely efficient; for example, we have demonstrated full recovery of an AES secret key from a Linux encrypted filesystem using just 800 analyzed encryptions, gathered within 65 milliseconds. A second side channel we observe is acoustic emanations: modern PCs produce unintentional acoustic signals that are highly correlated with processor activity, and reveal information about the calculation being performed (e.g., the secret moduli used during RSA decryption). Due to their efficiency and ubiquity, these newly recognized side-channel attacks form a threat in widespread practical circumstances, and thwart many traditional software security measures such as process separation, sandboxes and virtual machines.

תקציר

הגישה התאורטית לתורת ההצפנה מייצגת את המשתתפים במערכת, בין אם מורשים ובין אם לאו, כמכונות חישוב מופשטות עם ממשק מוגדר, ובוחנת את בטיחותם וסיבוכיותם במודל חישובי נוח כגון מודל דמוי-PC עם זיכרון-גישה-אקראית. דיסרטציה זו חוקרת מספר מקרים בהם המציאות חורגת מגבולות המודל, ובכך פותחת אפשרות לדרכים חדשות לשבירת צפנים.

חלקה הראשון של הדיסרטציה חוקר את העלות של פירוק מספרים לגורמיהם הראשוניים, ובעיקר את המקרה הפרטי של שבירת מערכת ההצפנה RSA עבור מפתחות בגודל נפוץ כגון 1024 ביטים. עד לאחרונה, משימה זו נחשבה לבלתי ישימה (עלותה נחזתה כטריליוני דולרים), בהתבסס על אקסטרפולציה המניחה מימוש מבוסס מחשבים טוריים דמויי PC. הראינו שהמצב שונה מהותית כאשר שוקלים גם ארכיטקטורות חומרה ייעודיות, הנעזרות באלגוריתמים ופרמטרים אשר נבחרו באופן מיטבי על פי שיקולים טכנולוגיים, ואשר חורגות מן המודל החישובי הסדרתי. בהתבסס על אלגוריתם Number Field Sieve (NFS) לפירוק לראשוניים, אנו מציגים ארכיטקטורות חומרה לשני השלבים העיקריים שבו – שלב הסינון ושלב האלגברה הלינארית. ניתוח מפורט מראה שעבור שבירת מפתחות RSA באורך 1024 ביטים, ניתן לממש את אלגוריתם NFS בעלות מעשית של מיליוני דולרים בודדים עבור תפוקה של פיצוח מפתח בשנה. הדבר מעלה חששות כבדים לבטיחותם של מפתחות אלו, אשר נמצאים בשימוש רחב ומומלצים בתקנים.

חלקה השני של הדיסרטציה עוסק בהפרה נוספת של ההפשטה החישובית: זליגת מידע ממערכות הצפנה דרך ערוצים צדדיים. אנו מציגים שני ערוצים כאלו. הערוץ ראשון הוא תחרות על זכרון המטמון במעבדים מודרניים; זו גוררת זליגת מידע אודות דפוסי גישה לזכרון, ואנו מראים דרכים לניצול זליגה זו לשם שבירת צפנים. ההתקפות החדשות ממומשות בתוכנה וישימות אפילו בתרחישים בהם למתקיף אין אף ערוץ תקשורת נומינלי עם תוכנת ההצפנה המותקפת. ההתקפות המוצגות יעילות ביותר: הדגמנו חשיפה מלאה של מפתח AES סודי מתוך מערכת קבצים מוצפנת של Linux תוך שימוש ב-800 הצפנות, דהיינו 65 מילישניות של מדידות, בלבד. ערוץ נוסף של זליגת מידע הוא זליגות אקוסטיות. מחשבים מודרניים, כך גילינו, פולטים שלא במתכוון רעשים המגלים מידע רב אודות החישוב המתבצע – לדוגמה, אודות מפתחות RSA הסודיים המשמשים לפענוח. בשל יעילותן וזמינות יישומן, ההתקפות שזיהינו מהוות איום בקשת רחבה של תרחישים מעשיים, ועוקפות אמצעי בטיחות מקובלים כגון מידור תהליכים ומכונות וירטואליות.

On the significance of this research

In a society that is becoming increasingly reliant on distributed communication networks, and an economy that is increasingly information-driven, the task of securely channeling information becomes as important as the task of creating it — just as irrigating a field of crop may require not merely the water but also an aqueduct to channel it. The Assyrians of Nineveh built masterful aqueducts to support their "exceedingly great city" [95]; contemporary cryptography builds secure channels of information to fulfill people's needs for trust, privacy and commerce. And as Nineveh fell to hydrological disaster [147], so does a modern peril lie in failure of cryptographic systems.

My research reveals several such imminent risks in well-established cryptographic and security settings. It has shaken the confidence in current deployment of RSA and AES, among the two most prominent cryptographic schemes, in such ubiquitous applications as e-commerce, banking, virtual private networks and encrypted disks. The techniques thus introduced have ramifications to many other cryptographic systems. Recognizing these hithero unknown risks, the computing industry responded by revising its practices and standards (e.g., those of the IEEE [88], Internet RFCs [159][215], USA government [152] and German government [175]) and researching mitigation (e.g, by Intel Corp. [36][37]).

My effective cache-based attacks have unexpectedly felled AES, an encryption algorithm chosen after years of exacting worldwide cryptographic scrutiny; this has spurred much research which extends the attacks and countermeasures described herein. My results on the cost of factoring (and hence the security of RSA) yield concrete estimates that, despite decades of investigation, are a million times lower than previous estimates, thereby showing the poignancy of this approach to custom-hardware cryptanalytic architectures. Numerous works, and a series of dedicated workshops, have since been fruitfully following this path.

Acknowledgments

Through this maiden voyage of research and discovery, my advisor Adi Shamir served as a compass, a guide and a partner. I am ever indebted for his wise advice, exacting mentorship and unfailing support, as well as for his admirable capacity for sharing his acumen and formidable knowledge. It has been a great privilege to learn the markings of a scientist from him.

I am indebted to my coauthors for our fruitful collaboration: Boaz Barak, Bruce Dodson, James Hughes, Willi Geiselmann, Wil Kortsmit, Paul Leyland, Moni Naor, Asaf Nussboim, Dag Arne Osvik, Ronen Shaltiel, Rainer Steinwandt and Jim Tomlinson — and in particular to Arjen K. Lenstra, for his perspective on computational number theory as well as on the finer aspects of scientific life.

The Weizmann Institute of Science has provided a wondrous environment for learning, cooperation and intellectual growth. I have greatly benefited from, and enjoyed, the conversations with the faculty and other residents of the department, and especially Moni Naor, Oded Goldreich, and Shafi Goldwasser. The institutes's Feinberg Graduate School, under Yosef Yarden and Ami Shalit, has judiciously monitored, supported and awarded my progress. I could not envisage a better place for a budding researcher.

Many other people have contributed to my research by comments, suggestions and discussions: Adi Akavia, Dan Bernstein, Ernie Brickell, Uriel Feige, Andrew Huang, Markus Jakobsson, Robert D. Silverman, Pankaj Rohatgi, Jean-Pierre Seifert, Thorsten Kleinjung, Eran Ofek, Michael Szydlo, David Tromer, Udi Wieder, Gideon Yuval, and numerous others. Dan Boneh, David Brumley, Jens Franke, Thorsten Kleinjung and Herman te Riele provided us with indispensable research data and code. National Instruments Israel donated lab equipment, and Nir Yaniv granted us use of the Nir Space Station studio.

Eli Biham, Shafi Goldwasser, Moni Naor and Omer Reingold graciously served on my interim and final examination committees.

Parts of my research was conducted while at Microsoft Research, hosted by Ramarathnam Venkatesan and Yacov Yacobi.

Back during my studies at the Technion, Erez Petrank and Eli Biham have first instilled in me the appreciation of computational complexity and cryptography; this I shall forever cherish.

My research was conducted, and this thesis written, using a variety of free (libre) software: the Linux kernel, GNU and KDE program suits, the Emacs, Kile and LyX, editors, the teTEX

LaTeX typesetting system, the GNU Multiple Precision Library, the Pari/GP and octave numerical/algebraic calculators, the Xfig, OpenOffice Impress and gnuplot drawing/plotting programs, the gcc compiler, the Perl interpreter, and numerous other programs from the the Fedora Core distribution. The OpenSSL and GnuPG cryptographic programs, as well as the Linux kernel, were also employed as test subjects for our cryptanalytic attacks.

I wish to thank my parents, Arie and Zahava, for their ever-present love and support, and for letting me take apart all those gadgets (even though they didn't always fit back together).

My gratitude extends to the many friends and family who helped and bore with me through recent years, and in particular to Yivsam and Zoya Azgad.

In love and appreciation of my wife Shlomit, this dissertation is dedicated to her:

Summary of contents

Contents

List of Figures

List of Tables

Part I

Hardware-based parallelization of the Number Field Sieve

The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic. It has engaged the industry and wisdom of ancient and modern geometers to such an extent that it would be superfluous to discuss the problem at length. [...]
Furthermore, the dignity of the science itself seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated.

— C. F. Gauss, 1801 [68, article 329]

Chapter 1

Introduction

Given any two numbers, we may by a simple and infallible process obtain their product, but it is quite another matter when a large number is given to determine its factors. Can the reader say what two numbers multiplied together will produce the number 8,616,460,799? I think it is unlikely that any one but myself will ever know; for they are two large prime numbers, and can only be rediscovered by trying in succession a long series of prime divisors until the right one be fallen upon.

— *W. S. Jevons, 1874 [94, page 141]*

1.1 Overview of Part I

Part I addresses the concrete cost of integer factorization and its effect on the security of the RSA cryptosystem. It proposes novel implementations of the Number Field Sieve factoring algorithm, using custom-built hardware devices that achieve massive parallelism, essentially for free. These designs feature an interplay between algorithmic, number-theoretical and technological aspects. By devising algorithms that take advantage of certain properties of the problem and of chip manufacturing technology, efficiency is increased by many orders of magnitude compared to previous proposals. For 1024-bit composites, the resulting cost is a few million US dollars — a millionth of previous predictions, and sufficiently practical to affect industry practice and standards (see §6.3).

The remainder of Chapter 1 surveys the state of the art in integer factorization and its cryptographic significance, and describes past approaches and pertinent works. It also recalls the pertinent details of the Number Field Sieve algorithm, focusing on its two computationally dominant stages, the sieving step and the linear algebra step.

Chapter 2 presents TWIRL, a novel special-purpose architecture for the NFS sieving step. This device is many orders of magnitude more cost-effective than previous proposals, and brings the

sieving step for 1024-bit integers to well within the practical realm: from trillions of US$ to mere millions

Chapter 3 addresses the NFS linear algebra step, and considers mesh-based hardware architectures for its realization. We begin by briefly evaluating a proposal by Bernstein, showing its practical deficiencies. We then proceed to suggest a different mesh-based algorithms and improve it in various ways, yielding far better scalability and, once again, reducing costs to within the practical realm — albeit with some technological reservations.

Chapter 4 describes an alternative special-purpose device for the NFS linear algebra step. Instead of a mesh, this architecture is based on a pipelined systolic architecture reminiscent of the TWIRL device. It resolves the technological hurdles of the previous approach, further reduces cost, and offers advantages in simulation and verification. We also suggest a highly efficient algorithmic fault detection scheme for the NFS linear algebra step.

Chapter 5 details our analysis of the NFS parameters that will arise when factoring 1024-bit and 768-bit composites, and demonstrate the pi falls one may encounter when following standard techniques. This analysis is necessary for the design and evaluation in preceding chapters. Our analysis indeed shows that the auxiliary steps of NFS can be kept at a cost lower than that of the two major steps addressed above — supporting the implications regarding the practicality of breaking 1024-bit RSA keys.

Chapter 6 summarizes and discusses the results of Part I, and briefly surveys recent progress following our publications.

1.2 Integer factorization

The problem of finding the prime factors of integers is one of the oldest problems in mathematics. Part of its appeal, as put forth by Gauss (see page 18), lies in the elegance and simplicity of the problem's definition: given an integer n, find integers $p, q \neq \pm 1$ such that $n = pq$. Despite its simple statement, this has proved to be one of the notoriously difficult problems in computational number theory.[1]

In 1977, the factoring problem gained great significance in cryptography with the introduction of the RSA cryptosystem [176], the first public-key encryption scheme and signature scheme with (conjectured) super-polynomial security. In this cryptosystem, the secret key is a pair of large primes p, q and the public key is their product $n = pq$; its security thus relies on the hardness of factorization.[2] Subsequently, several other public-key cryptosystems based on the hardness of factoring were proposed (e.g., those of Rabin [172] and Paillier [164]), as well as

[1] For the related problem of testing whether a given integer is prime, probabilistic polynomial-time algorithms are known for several decades [141][199][173], and the problem was completely solved by the deterministic polynomial-time algorithm of Agrawal et al. [8].

[2] It is not known whether hardness of factorization *suffices* for the security for RSA, but for appropriate parameters this is conjectured to be true.

other cryptographic primitives such as the Blum-Blum-Shub pseudorandom generator [29] and the VSH hash function [46]. To date, the hardness of factorization is one of the few plausible hardness assumptions on which to base public-key cryptography [81], and specifically one of the very few candidate one-way trapdoor permutations [80]. As such, it relates to deep open problems in complexity theory and the foundations of cryptography (e.g., as posed by Impagliazzo[3] [90]).

Beyond the theoretical interest, there is a practical motivation for studying the concrete hardness of integer factorization. RSA, being the most commonly deployed public-key cryptosystem, is ubiquitous in such diverse contexts as secure web sites employing SSL/TLS, S/MIME encrypted e-mail, various e-commerce and banking applications, and code authentication (e.g., web applets, software updates, and game console titles). Hence, all of these presently assume the hardness of integer factorization for their security. As shown in this dissertation, these assumptions are far from warranted.

Various non-trivial algorithms have been devised for integer factorization, starting with observations by Fermat in 1643 [54]. Recent decades have seen especially fruitful, with a cascade of discovered algorithms. One line of development, seeking algorithms whose complexity depends on the size of the composite n, includes Dixon's algorithm, the Continued Fraction method, the Morrison-Brillhart approach, the Quadratic Sieve, and ultimately the Number Field Sieve — along with their many variants. The other line of development is concerned with algorithms whose complexity (for partial factorization, i.e., finding any nontrivial factor) depends only on the size of the smallest factor of n; these include Pollard rho, Pollard $p-1$, the ultimately the Elliptic Curve method. See the surveys of A. K. Lenstra [125] for a comprehensive survey of modern algorithms, H. C. Williams at al. [218] for a detailed account of historical factorization methods circa 1750–1950, and L. E. Dickson [55] for older methods.

We do not know the true computational complexity of factoring integers drawn from the distributions of interest. The associated decision problem ("does n have a factor smaller than x?") has efficiently-verifiable witnesses (namely the factorization of n), and is thus in **NP** $\cap$ co-**NP**; consequentially it is believed that factorization is not **NP**-hard, as that would imply a collapse of the polynomial hierarchy. On quantum computers, the problem can be solved in polynomial time via Shor's algorithm [192] [193], but to date little progress has been achieved towards an empirical quantum realization of this algorithm and verification of its scalability.[4] While the problem is widely believed to be hard, this confidence relies solely on the absence of an efficient factoring algorithm despite centuries of research.

The best algorithm known for factoring large integers of a general form is the Number Field Sieve.

[3] Impagliazzo [90, §2.5] writes: "*Currently, all known secure public key cryptosystems are based on variants of RSA, Rabin, and Diffie-Hellman cryptosystems. If an efficient way of factoring integers and solving discrete logarithms became known, then not only would the popular public key cryptosystems be broken, but there would be no candidate for a secure public-key cryptosystem, or any real methodology for coming up with such a candidate.*" This largely still holds today, with Diffie-Hellman extended to elliptic curves and with the notable addition of lattice-based cryptosystem such as GGH [82], Ajtai-Dwork [9] and NTRU [85] (see [174] for a partial survey and further pointers). Alas, the latter do not presently enjoy the same level of confidence as factoring.

[4] The only such published experiment used a 7-qubit quantum computer to factor $15 = 3 \cdot 5$, using inherently non-scalable nuclear magnetic resonance techniques [209].

The (heuristic) asymptotic time and space complexities of NFS are of the form

$$e^{(c+o(1))\ (\log n)^{1/3}\ (\log\log n)^{2/3}}$$

where n is the composite being factored, and the constant c depends on the complexity measure and the variant of the algorithm (see §5.2.1.4). While this algorithm has been fruitfully employed to many a challenging factoring problems (the record stands at 663-bit bits [15][63]), it is not thought to be capable of tackling the factorization problems that arise in practical cryptography, such as factoring 1024-bit RSA keys.

The remainder of this chapter will survey the Number Field Sieve algorithm, its implementations, and concrete factorization experiments and challenges.

1.3 Empirical hardness of factorization

1.3.1 Challenges and past experiments

Factoring integers with current techniques poses significant algorithmic and engineering challenges, and involves considerable uncertainty due to aspects of the algorithm for which analysis is lacking or not tight. Over the past decades, this has led to numerous factorization experiments which employed the state-of-the-art in factoring algorithms and computer technology (of their time) to tackle sample challenges. For composites of a special form suitable to the Special Number Field Sieve, there exist natural challenges such as Fermat numbers [128]. For integers of a general form, which are of greater cryptographic usefulness and which form our focus, it is common to choose targets from the "RSA challenge" list of composites. This list was published by RSA Data Security, Inc. in 1991 and revised in 2001 [177], and consists of essentially randomly-drawn RSA composites (i.e., integers with two prime factors of similar size) whose factorization was not known at the time of publication.[5] The hardness of factoring the RSA challenge composites is widely taken to be representative of the general factoring task for RSA composites of comparable size.

To date, the largest completed experiment (for integers of a general form) has factored the 663-bit composite RSA-200 by Franke et al. [15][63]. Previous records are the 640-bit composite RSA-640 [16], the 576-bit composite RSA-576 [62] and the 530-bit composite RSA-160 [17], by the same group. However, little information has been released about these recent experiments. The best-documented NFS factorization experiment is that of the 512-bit composite RSA-155 [44][43]; we shall at various points rely on empirical information gleaned at that experiment (e.g., in Chapter 5). Prior to that, various smaller composites such as RSA-140 [201] were factored via NFS and its predecessors.[6] More recently, Aoki et al. [12] revisited smaller remaining challenges such as RSA-150 and provided statistics about their factorization via NFS.

[5] The process of generating the composites involved knowledge of their factorization, but according to [177], this information (and all potential side-channel information) has been contained and destroyed.

[6] See e.g. [35], [44, Table 1] or [197] for a list of factoring records since 1970.

1.3.2 1024-bit RSA and its importance

The present "holy grail" for empirical factorization are 1024-bit composites, exemplified by the RSA-1024 challenge number. RSA keys of this size are very widely deployed, including banking services and the overwhelming majority of secure web sites (including e-commerce hubs such as Amazon and eBay).

Such composites have been posited to be impractical to factor (by conventional means) until 2015 at least. According to Silverman's extrapolation using NFS asymptotics [197] (adjusted to prices circa 2005), completing such a factorization in 1 year would require roughly US$ 10^{12} worth of PC hardware. Brent [35] showed that a high-level extrapolation from past factoring records suggests that 1024-bit composites will be factored no sooner than 2018. Accordingly, until recently a NIST draft guideline [151] suggests 1024-bit RSA keys can be considered secure until 2015 (this was revised [152] to 2010 after the publication of our research, and several industry standards were likewise affected; see §6.3).

Industry leaders appear even more optimistic about the security of such keys. For example, VeriSign Inc., the largest commercial certificate authority, has issued multiple 1024-bit root certificates that are set to expire in 2028. Even RSA Security Inc. distributes a 1024-bit certificate authority keys valid until 2026. These certificates are built into, and trusted by, all major web browsers.[7] If these certificates are compromised prematurely, then browser SSL certificates will become untrustworthy during the many years it would take to revoke all such certificates from users' computers.

Given this motivation, our main focus in the next chapters will be on the cost of factoring 1024-bit composites, and RSA-1024 specifically. To alleviate suspicion, we have duplicated some of the analysis in Chapter 5 using randomly drawn 1024-bit RSA composites instead of RSA-1024; no significant difference in the behavior was detected.

1.3.3 768-bit RSA and its importance

As an intermediate and more accessible goal, and following prior literature (e.g., [197]), we shall at times also consider RSA-768 challenge number [177], as representative of 768-bit RSA composites. RSA keys of this size are of lesser direct practical interest, as they have been mostly phased out, but successfully factoring them is expected to enhance our understanding of the scalability and behavior of the NFS algorithm.

[7] For example, certificates named "`VeriSign Class {1,2,3,4} Primary CA`" in Internet Explorer 6, and likewise certificates with "`OU=VeriSign Class {1,2,3} Public Primary Certificate Authority`" and "`OU=RSA Security 1024 V3`" in Firefox 2.0.

1.4 Cost measures

Algorithms are traditionally analyzed in terms of their time complexity (number of operations in an abstract model, or concrete running time) and space complexity (amount of storage in some abstract model, or circuit size in some abstract model, or the concrete construction cost of a device). In our focus on very large scale cryptanalytic problems, however, it is often more useful to consider the product of these two costs, known as "AT cost" (for area×time) in VLSI design. We shall refer to this cost measure as *throughput cost*, to stress its motivation in measuring the equipment cost per unit problem-solving throughput. We shall usually consider it in concrete terms of construction cost and actual running time, e.g., measured in US$\$\times$years.

It appears that throughput cost is indeed appropriate when a large number of problems must be solved during some long period of time while minimizing total expenses. In particular, it is conveniently oblivious to time-space tradeoffs that do not affect that total cost. One should, however, be wary of non-discriminating use in security assessments: an adversary in possession of plans for a device that breaks one key per year and costs $1M to construct clearly forms a greater menace, compared to an adversary with plans for a device that costs $1 but requires 1M years of sequential computation per key.

The use of this cost measure in the context of integer factorization was brought to the spotlight by Bernstein [22]. An asymptotic treatment in an abstract model, in several contexts, was given by Wiener [217].[8]

When evaluating special-purpose, custom-built cryptanalytic devices, there are additional costs to consider. Beyond the marginal cost of constructing each device, there are typically significant *Non Recurring Engineering* (NRE) costs for setting up the production line of the device, e.g., for development, creation and verification of the lithographic masks for a VLSI process. There are also costs involved with power and cooling, physical housing, and maintenance. While these are not our focus, we shall occasionally refer to these considerations as well.

The power and cooling costs reflect the specifics of present technology; in principle any computation can be efficiently implemented using reversible gates and arbitrarily low *total* energy expenditure. However, there exists a physical lower bound on the throughput cost of any fixed algorithm, due to a trade-off between speed and total energy.[9]

[8] Notably, Wiener's bounds assume a 3-dimensional circuit whereas the devices discussed here all rely on 2-dimensional VLSI circuits. If considered asymptotically and adapted to 3 dimensions in the natural manner, our devices match Wiener's lower and upper bounds.

[9] Lloyd [134] shows an upper bound of $5.43 \cdot 10^{50}$ logical operations per second per kilogram of mass, due to (a variant of) Heisenberg's uncertainty principle. It follows that when performing a given number of operations, the product of mass (hence energy and cost) and time cannot be arbitrarily low.

1.5 The Number Field Sieve algorithm

1.5.1 Background

The Number Field Sieve algorithm was first proposed by J. M. Pollard in 1988, for composites of a special form (this variant is often referred to as the *Special Number Field Sieve*). It was subsequently implemented, improved and extended to the general case in a series of works by A. K. Lenstra, H. W. Lenstra, M. Manasse, C. Pomerance, J. Buhler, L. M. Adleman, P. Montgomery, D. Coppersmith and others. The resulting algorithm is known as the *General Number Field Sieve*, often abbreviated *Number Field Sieve* (NFS). Many additional incremental improvements were suggested during the subsequent two decades. See [129] for an account of the dawn of the NFS and reprints of the seminal works, and [168] for an introduction and historical context.

A full description of the Number Field Sieve taking advantage of all published techniques and variations is rather non-trivial, and to our knowledge no such implementation presently exists. This section describes the parts of the Number Field Sieve algorithm which are relevant to this dissertation. Throughout the discussion, we focus on a common variant of the General Number Field Sieve, which has been employed in many large-scale factoring experiments.

1.5.2 Notation

The following is our basic terminology and notation when discussing the Number Field Sieve. Each of the subsequent chapters introduces its own additional local notation, for discussing architectural parameters (Chapter 2, Chapter 3 and Chapter 4) or number-theoretical functions and additional NFS parameters (Chapter 5). See the Index of notation on page 195.

The number of primes smaller than or equal to x is denoted by $\pi(x)$.

Following Knuth and Pardo [107], we denote the prime factors of an integer $z > 1$ by $z_{(1)}$, $z_{(2)}$, $z_{(3)}$, $\dots$ where $|z| = z_{(1)} \cdot z_{(2)} \cdot z_{(3)} \cdots$ and $z_{(1)} \geq z_{(2)} \geq z_{(3)} \geq \cdots$ are prime or 1.

Let $x \in \mathbb{Z}$, $U > 0$ (a *smoothness bound*), $V > 0$ (a *large prime bound*) and ℓ a non-negative integer. Define the following properties of integers (see also §5.2.3):

An integer z is called *U-smooth* if $z_{(1)} \leq U$, i.e., if all its prime factors are at most U.

More generally, z is called *(U, V, ℓ)-semismooth* if $z_{(\ell)} \leq V$ and $z_{(\ell+1)} \leq U$, i.e., all its prime factors are smaller than or equal to U except for at most that are smaller than or equal to V; the latter are referred to as *large primes*.

Lastly, z is called *strictly (U, V, ℓ)-semismooth* if $U < z_{(\ell)} \leq V$ and $z_{(\ell+1)} \leq U$, i.e., all its prime factors are smaller than or equal to U except for *exactly* ℓ that are smaller than or equal to V.

Let $\mathbb{Z}_n \stackrel{\text{def}}{=} \mathbb{Z}/(n\mathbb{Z})\mathbb{Z}_n$ denote the ring of integers modulo n. For a field $\mathbb{F}$, $\mathbb{F}^{n \times m}$ denotes the ring of $n \times m$ matrices over $\mathbb{F}$ and $\mathbb{F}^n$ denotes the vector space of dimension n over $\mathbb{F}$. Matrices shall be denotes by capital Latin letters (e.g., A). Column vectors are written as $\vec{v}$, and vector

transposition is denoted $\vec{v}^{\mathrm{T}}$. For a vector $\vec{v}$, both v_i and $(\vec{v})_i$ denote its i-th element of $\vec{v}$; but $\vec{v}_i$ is just some other vector.

The natural logarithm is denoted by $\ln(x)$ and the logarithm to base 2 by $\lg(x)$. Let $\exp(x) = e^x$, and let $x\mathrm{E}y$ denote $x \cdot 10^y$ (e.g., 1.2E3). The cardinality of a set S is denoted $|S|$. We use the standard "Big-O" notations $O(x)$, $o(x)$ and $\tilde{O}(x)$, where the latter disregards polylogarithmic factors.

In our concrete cost estimates, where not specified otherwise the "$" currency is US$ circa 2004. In several places we will use notation such as $\langle\!\langle x \rangle\!\rangle$ to denote concrete parameters or costs; these will be explained in the relevant context.

1.5.3 Overview of NFS

Let n be the number to be factored. We begin by choosing two irreducible polynomials, $f(X), g(X) \in \mathbb{Z}[X]$, that have no common root over $\mathbb{Z}$, but share a common root m modulo n:

$$f(m) \equiv g(m) \equiv 0 \pmod{n} \ .$$

Let α be a complex root of f, and consider the ring $\mathbb{Z}[\alpha]$ of polynomials in α with integers coefficients (it is homomorphic to $\mathbb{Z}[X]/(f(X))$). Since $f(\alpha) = 0$ in $\mathbb{Z}(\alpha)$ and $f(m) = 0$ in Z_n, the mapping $\alpha \mapsto m$ induces a ring homomorphism $\phi : \mathbb{Z}[\alpha] \to \mathbb{Z}_n$. Analogously, letting β be a complex root of g, the mapping $\beta \mapsto m$ induces a ring homomorphism $\psi : \mathbb{Z}[\beta] \to \mathbb{Z}_n$.

Suppose we were able to find a finite set of coprime integers, $\mathcal{T} \subset \mathbb{Z} \times \mathbb{Z}$, and elements $\gamma \in \mathbb{Z}[\alpha]$ and $\delta \in \mathbb{Z}[\beta]$, such that

$$\prod_{(a,b)\in\mathcal{T}} (a - \alpha b) = \gamma^2 \quad , \qquad \prod_{(a,b)\in\mathcal{T}} (a - \beta b) = \delta^2 \ . \tag{1.1}$$

Applying the ring homomorphisms ϕ and ψ on the left and right equations respectively, this yields:

$$\prod_{(a,b)\in\mathcal{T}} (a - mb) = \phi(\gamma)^2 \quad , \qquad \prod_{(a,b)\in\mathcal{T}} (a - mb) = \psi(\delta)^2$$

over $\mathbb{Z}_n$. Denoting $x = \phi(\gamma)$ and $y = \psi(\delta)$, we have thus obtained the congruence

$$x^2 \equiv y^2 \pmod{n} \ .$$

If n factors as $n = pq$ with $p, q \neq \pm 1$ (not necessarily prime) and the choice of $\mathcal{T}$ was sufficiently random (in a heuristic sense), then with probability at least 1/2 one of these holds: $x = y \pmod{p}$, $x = -y \pmod{q}$; or $x = -y \pmod{p}$, $x = y \pmod{q}$. In either of these cases, $\gcd(n, x-y)$ is a nontrivial factor of n; otherwise we retry with a different $\mathcal{T}$. Finally, if $\gcd(n, x-y)$ or $n/\gcd(n, x-y)$ are not prime, they can be factored recursively.

The crux of this algorithm is finding a set $\mathcal{T}$ which fulfills (1.1), i.e., for which $\prod_{(a,b)\in\mathcal{T}}(a-\alpha b)$ and $\prod_{(a,b)\in\mathcal{T}}(a-\beta b)$ are both squares in the respective rings.

Following the approach of Morrison-Brillhart and Dixon (see [129]), this problem is reduced to two main steps, a *sieving step* and a *linear algebra step*. These two steps dominate the computational complexity of the NFS, and as such form our focus; we shall discuss them in detail in subsequent sections. The overall procedure for finding $\mathcal{T}$ is as follows.

Fix rational smoothness and semismoothness bounds $U_{\mathbf{r}}$ and $V_{\mathbf{r}}$ respectively, with $U_{\mathbf{r}} \leq V_{\mathbf{r}}$. Likewise, fix algebraic smoothness and semismoothness bounds $U_{\mathbf{a}}$ and $V_{\mathbf{a}}$, with $U_{\mathbf{a}} \leq V_{\mathbf{a}}$. Fix the number of *large primes*: $\ell_{\mathbf{a}}$ on the rational side and $\ell_{\mathbf{r}}$ on the algebraic side.

Norms. Assume for simplicity that f and g are monic and g is linear.[10] For coprime integers (a,b) with $b \neq 0$, we define the *rational norm* and *algebraic norm* as

$$N_{\mathbf{r}}(a,b) \stackrel{\text{def}}{=} |a-bm| \quad , \quad N_{\mathbf{a}}(a,b) \stackrel{\text{def}}{=} |b^d f(a/b)| \ .$$

Sieving. In the *sieving step* (also called *relation collection*) we look for *relations*: pairs of coprime integers (a,b) with $b>0$ such that $N_{\mathbf{r}}(a,b)$ is $(U_{\mathbf{r}},V_{\mathbf{r}},\ell_{\mathbf{r}})$-semismooth and $N_{\mathbf{a}}(a,b)$ is $(U_{\mathbf{a}},V_{\mathbf{a}},\ell_{\mathbf{a}})$-semismooth. If $N_{\mathbf{r}}(a,b)$ is $U_{\mathbf{r}}$-smooth and $N_{\mathbf{a}}(a,b)$ is $U_{\mathbf{a}}$-smooth, the relation is referred to as a *full relation*, otherwise it is called a *partial relation*.

Approximately $\pi(\min\{U_{\mathbf{r}},U_{\mathbf{a}}\})/d!$ full relations are *free*, namely one for each prime $p \leq \min(U_{\mathbf{r}},U_{\mathbf{a}})$ such that f has d roots modulo p (see [129]).

For non-negative integers $\ell'_{\mathbf{r}}$ and $\ell'_{\mathbf{a}}$, a non-free relation (a,b) for which $N_{\mathbf{r}}(a,b)$ is strictly $(U_{\mathbf{r}},V_{\mathbf{r}},\ell'_{\mathbf{r}})$-semismooth and $N_{\mathbf{a}}(a,b)$ is strictly $(U_{\mathbf{a}},V_{\mathbf{a}},\ell'_{\mathbf{a}})$-semismooth is called an $(\ell'_{\mathbf{r}},\ell'_{\mathbf{a}})$-*partial relation*. We use the standard abbreviations *ff* for (0,0)-partial ("full,full") relations, *fp* for (0,1)-partial ("full,partial") relations, *fp* for (1,0)-partial relations and *pp* for (1,1)-partial relations.

On the rational side, the sieving step involves sieving with the *rational factor base*: the set of primes $p \leq U_{\mathbf{r}}$, whose cardinality is $\pi(U_{\mathbf{r}})$. On the algebraic side, sieving involves an *algebraic factor base*: the set of pairs (p,r) with $p \leq U_{\mathbf{a}}$ prime and $f(r) \equiv 0 \bmod p$, whose cardinality $\approx \pi(U_{\mathbf{a}})$. These factor bases and parameters determine the progressions and thresholds in the sieving procedure detailed in §1.6

The part of the (a,b)-plane where relations are sought, called the *sieving region*, is

$$\mathcal{S} \stackrel{\text{def}}{=} \{(a,b) \ | \ -A < a \leq A, \ 0 < b \leq B\}$$

for appropriately chosen bounds $A,B>0$.[11] The ratio $\omega \stackrel{\text{def}}{=} A/B$ is called the *skewness ratio* of the sieving region. We denote the size of the sieving region by $S \stackrel{\text{def}}{=} |\mathcal{S}| = 2AB$. For any fixed

[10] For general $g(x)$ we redefine $N_{\mathbf{r}}(a,b) \stackrel{\text{def}}{=} |b^{\deg(g)} \cdot g(a/b)|$ (the above is a special case); however this general case entails some complications that are not relevant in our context, so for simplicity we will mostly ignore this extension, and use it only in §5.3 for the special case of non-monic linear $g(X)$ and, implicitly, when referring to Coppersmith's variant in §5.2.1.3 and §5.4.2.3.

[11] Such a rectangular sieving region is in general not optimal: a carefully chosen and somewhat smaller region, taking into account the real roots of f, can yield the same number of relations (see [196]). For our yield computations, the effect is minor.

$0 < b \leq B$, the subset

$$\{(a, b) \mid -A < a \leq A\}$$

is called a *sieve line*, and the *sieve line width* is $R \stackrel{\text{def}}{=} 2A$.

Cycles. A *cycle* is a set C of relations such that $\prod_{(a,b)\in C} N_{\mathbf{r}}(a, b)$ is a square times a $U_{\mathbf{r}}$-smooth number and, simultaneously, $\prod_{(a,b)\in C} N_{\mathbf{a}}(a, b)$ is a square times a $U_{\mathbf{a}}$-smooth number[12]. For example, a full relation forms a cycle of length 1. Two (1,0)-partial relations whose rational norms share a large prime can be combined into a cycle of length 2. Similarly, for two (0,1)-partial relations (a_1, b_1) and (a_2, b_2) whose algebraic norms share the large prime p, a length 2 cycle follows if the relations correspond to the same root of f mod p, i.e., if $a_1/b_1 \equiv a_2/b_2 \bmod p$. Longer cycles may be built by pairing matching rational large primes or matching algebraic large primes with corresponding roots.

Combining relations into cycles. We say that a set of cycles is *independent* if the characteristic vectors of the cycles (as sets of relations) are linearly independent. Various techniques can be used to construct independent cycles from relations; see [43] for an account. The number of constructable independent cycles of small length clearly grows as additional relations are found, but the expected growth rate is imperfectly understood, especially when many large primes are allowed (large $\ell_{\mathbf{a}}$ and $\ell_{\mathbf{r}}$).[13] In Chapter 5 we shall rely on some heuristic and experimental estimates to address this.

Combining cycles into $\mathcal{T}$. For a cycle C, define $\sigma_C = \prod_{(a,b)\in C}(a - \alpha b)$. In the number field $\mathbb{Q}(\alpha)$, the principal ideal generated by σ_C can be uniquely factored into a product of prime ideals: $\sigma_C = \prod I_i^{q_{C,i}}$, where I_i are the prime ideals in $\mathbb{Q}(\alpha)$ (in arbitrary enumeration) and the $q_{C,i}$ are integers. This induces an infinite exponent vector $\vec{q}_C = (q_{C,1}, q_{C,2}, \ldots)$. Similarly, for any set $\mathcal{T}'$ of cycles, the product $\sigma_{\mathcal{T}} = \prod_{C\in\mathcal{T}'} \sigma_C$ has an exponent vector $\vec{q}_{\mathcal{T}} = \prod_{C\in\mathcal{T}'} \vec{q}_C$. By unique factorization into prime ideals, $\sigma_{\mathcal{T}}$ is a square iff all the elements of $\vec{q}_{\mathcal{T}}$ are even. And if $\sigma_{\mathcal{T}}$ is a square then the set $\mathcal{T} = \bigcup_{C\in\mathcal{T}'} C$ (as a union with multiplicity), fulfills the left side of (1.1). It follows that, given many cycles, we can try fulfilling (1.1) by computing their exponent vectors and finding a linear combination whose sum is 0 modulo 2.

Here a crucial observation is made, in order to step from number fields down to mere integers: the factorization of the ideal generated by σ_C into prime ideals of $\mathbb{Q}(\alpha)$ is closely related to factorization of the integer $x_C = \prod_{(a,b)\in C} N_{\mathbf{a}}(a, b)$ into prime numbers. This follows from the fact that $N_{\mathbf{a}}(a, b)$, as defined above, is the norm of $a - \alpha b$ in the number field, and has useful properties such as multiplicativity. Very roughly speaking, each prime ideal in the factorization of the ideal generated by σ_C is associated with some prime factor of x_C. Thus, instead of the exponent vectors $\vec{q}_C$, we can consider the exponent vectors $\vec{r}_C = (r_{C,1}, r_{C,2}, \ldots)$ defined by

[12] The condition on $\prod_{(a,b)\in C} N_{\mathbf{a}}(a, b)$ is slightly more involved, but this is inconsequential in our context; see [129] for the exact characterization.

[13] For details see Lenstra and Dodson's experiment, Lambert's partial analysis [113], and Cavallar's methods and results for RSA-155 [43].

factorization over the integers: $x_C = \prod {p_i}^{r_{C,i}}$, where p_i are the prime numbers (in increasing order) and the $r_{C,i}$ are integers.[14]

For each cycle C found earlier, we know the exponent vector $\vec{r}_C$. We wish to find a linear combination of these exponent vectors with all-even entries. By definition of a cycle, we know that all but the first $\pi(U_{\mathbf{a}})$ entries of every $\vec{r}_C$ are even, so we only have to deal with finite-length truncated vectors containing $\pi(U_{\mathbf{a}})$ entries.

Everything that has been said about also holds on the rational side, i.e., for the norm $N_{\mathbf{r}}(a,b)$ and the number field $\mathbb{Q}(\beta)$. Since we wish to satisfy both terms in (1.1), our task thus boils down to constructing a concatenated exponent vectors (of dimension $\pi(U_{\mathbf{r}})+\pi(U_{\mathbf{a}})$) for each cycle, and then finding a linear relation modulo 2 among these cycles.

Recall that $\pi(\min\{U_{\mathbf{r}},U_{\mathbf{a}}\})/d!$ full relations were obtained for free, so once sieving has created more than $\Pi(U_{\mathbf{r}},U_{\mathbf{a}}) \stackrel{\text{def}}{=} \pi(U_{\mathbf{r}})+\pi(U_{\mathbf{a}})-\pi(\min\{U_{\mathbf{r}},U_{\mathbf{a}}\})/d!$ (independent) cycles, we are guaranteed that a linear relation modulo 2 will exist among the corresponding vectors. Thus, the purpose of the sieving step is to find enough relations so that we can construct approximately $\Pi(U_{\mathbf{r}},U_{\mathbf{a}})$ independent cycles.

Linear algebra. In the *linear algebra* step (also called the *matrix step*), we take the exponent vectors (of dimension $\pi(U_{\mathbf{r}})+\pi(U_{\mathbf{a}})$) corresponding to $\Pi(U_{\mathbf{r}},U_{\mathbf{a}})$ independent cycles, add vectors corresponding to the free relations, and form a matrix A.[15] We then seek an nonzero element in the kernel of A (i.e., a linear relation). The result is the characteristic vector of a set $\mathcal{T}$ fulfilling (1.1), from which the factorization follows with probability at least 1/2. By finding several independent kernel elements of A, and under heuristic assumptions about the random-looking properties of the cycles, we can complete the factorization with overwhelming probability.

Choice of polynomials. The polynomials f and g are chosen as to maximize the number of $N_{\mathbf{r}}(a,b)$ and $N_{\mathbf{a}}(a,b)$ in the sieving region that satisfy the semi-smoothness conditions. This is determined mainly the size of the values assumed by $N_{\mathbf{r}}(a,b)$ and $N_{\mathbf{a}}(a,b)$ (smaller numbers are more likely to be semi-smooth), and by their "root properties" (see [146]). The choice of polynomials f and g can be done by a variety of methods. The original proposal (see [129]) was to pick m of suitable size (namely $m \approx n^{1/(d+1)}$ where $d \approx (\log n/\log\log n)^{1/3}$ is the desired degree of f), let $g(X) = X - m$, and let the coefficients of $f(X)$ be the base-m representation of n. Improved techniques were devised by Montgomery and Murphy (see [143], [145], [146]). At present the best results are achieved by the unpublished method of Kleinjung [64]; see further discussion and concrete examples in Chapter 5.

Most methods for selecting polynomials create g which is linear, in which case β (the primitive root of g) is rational and thus so are all elements of $\mathbb{Z}[\beta]$. We shall henceforth assume that f is of arbitrary degree while g is linear. We shall also follow the convention of referring to operations

[14] This account is greatly simplified, and complications abound; see [129] for the full details. The omitted complications do not significantly affect the subsequent discussion.

[15] Alas, the common NFS notations collide here: A denotes both the matrix and the boundary of sieving lines (an integer). The two are easily disambiguated by context, so we adhere to both conventions.

involving g and $\mathbb{Z}[\beta]$ as happening on the *rational side*, as opposed to operations involving f and $\mathbb{Z}[\alpha]$, which are said to happen on the *algebraic side*. While the two sides fulfill the same role in principle, the higher degree of f and the potentially complicated structure of $\mathbb{Z}[\alpha]$ on the algebraic side (see [39]) entail some differences in parametrization and choice of sub-algorithms, as shall be seen later.

Computing algebraic square roots. Given a set $\mathcal{T}$ which fulfills (1.1) for some algebraic numbers γ^2 and δ^2, one can efficiently compute the square roots γ and δ by employing methods of Couveignes [50], Montgomery and Nguyen [155].

Cofactor factorization and candidate testing. When multiple large primes are used ($\ell'_{\mathbf{r}} \geq 2$ or $\ell'_{\mathbf{a}} \geq 2$), the basic sieve algorithm §1.6 produces some false positives. It tests only for the presence of a large smooth factor of $N_{\mathbf{r}}(a, b)$ or $N_{\mathbf{a}}(a, b)$, but does not verify that the remaining factors ("cofactors") satisfy the semi-smoothness conditions. Thus, candidate (a, b) values that passed the basic sieve undergo further filtering via suitable small-scale factoring algorithms such as the Elliptic Curve Method (see §5.2.4.3 for additional details). Moreover, we need to factor the smooth parts as well, in order to construct the exponent vectors (if not provided by the sieving step in full) and to check the results of the possibly-unreliable sieving step.

1.5.4 NFS for discrete logarithms

Beyond integer factorization, another computational problem of great cryptographic significance is that of computing discrete logarithms in certain finite groups, such as $\mathbb{Z}_p^*$ (the multiplicative group of GF(p)). The ideas underlying NFS have been carried over to this problem, yielding an analogous algorithm with similar asymptotic behavior. The Number Field Sieve for discrete logarithms was first proposed by Gordon [56], and improved by Schirokauer [182]; see [156] for a survey.

The major computational steps in this algorithm are qualitatively similar to those of the NFS factorization algorithm. One significant difference is that in the linear algebra step, all operations are done over GF(q) instead of GF(2), where q is the order of the group. In typical cryptographic applications, q is at least 1024-bits long. Field operations and storage are thus significantly more expensive than in the case of factoring of similarly-sized composites.

1.6 The NFS sieving step

1.6.1 The task

The first major step of the NFS algorithm, which dominates the cost in practice, is the sieving step.[16] In a simplified form and after appropriate reductions [129], the sieving problem is as

[16] Asymptotically the cost of the two steps can be balanced by appropriate choice of trade-off parameters, but as argued in [131], this property does not appear to hold for the problem parameters of interest.

follows.[17]

The inputs of the sieving problem are $R \in \mathbb{Z}$ (*sieve line width*), $T > 0$ (*threshold*) and a set of pairs (p_i, r_i) where the p_i are the prime numbers smaller than some *factor base bound* U. There is, on average, one such pair per prime smaller than U, and thus roughly $B/\ln B$ pairs in total. Each pair (p_i, r_i) corresponds to an arithmetic progression:

$$P_i \stackrel{\text{def}}{=} \{a : a \equiv r_i \pmod{p_i}\}$$

We are interested in identifying the sieve locations, i.e., integers $a \in \{0, \ldots, R-1\}$, that are members of many progressions P_i with large p_i:

$$\lambda(a) > T \text{ where } \lambda(a) \stackrel{\text{def}}{=} \sum_{i:a \in P_i} \log_c p_i$$

for some small constant c.

Since we only care about the total yield of relations, the sieving step can tolerate a small fraction (e.g., a few percents) of lost relations with little effect — a similarly moderate increase in the sieving area will fulfill the quota.[18] It is also permissible to have (infrequent) false positives, since these will be filtered out during candidate testing. This flexibility is very useful for efficiency, since it relaxes constraints on accuracy and fault-tolerance (unlike the strict correctness requirements in the matrix step below). For example, we round all logarithms to the nearest integer (hence the non-arbitrary constant c).

Out of the R sieve locations, only $(6/\pi^2)R$ on average are potentially useful and the rest can be ignored (due to the requirement, in §1.5.3, that a and b are coprime).

In the NFS sieving step we have two types of sieves: *rational* and *algebraic*, corresponding to the polynomials f and g in the higher level of the NFS algorithm (see §1.5.3). Both sieving problems are of the above form, but differ in their factor base bounds ($U_{\mathbf{r}}$ vs. $U_{\mathbf{a}}$), threshold T and basis of logarithm c (typically $U_{\mathbf{a}} \gg U_{\mathbf{r}}$ so the algebraic side entails a higher computational load.[19]). We need to handle B sieve lines, and for sieve line both sieves are applied, so there are $2B$ sieving instances overall.

For each sieve line, each value a that passes the threshold in both sieves implies a *candidate*. Each candidate undergoes additional tests, for which it is beneficial to also know the set $\{i : a \in P_i\}$ (for each sieve separately). The most expensive part of these tests is *cofactor factorization*, which involves factoring medium-sized integers.[20] The candidates that pass the tests are called *relations*. The output of the relation collection step is the list of relations and their corresponding $\{i : a \in P_i\}$ sets. Our goal is to find a certain number of relations, and the parameters are chosen accordingly a priori.

[17] For simplicity we assume line sieving [129]. The description also applies to one (suboptimal) approach to lattice sieving. At one point (namely §2.3.8), we thus exceed the present model in order to better address lattice sieving.)

[18] This is true as long as the parameters are sufficiently close to optimum; see §5.2.2.2 and §5.5.1.

[19] In TWIRL the rational sieve dominates the cost, due to these of cascaded sieves.

[20] We assume use of the "2+2 large primes" variant of the NFS [129, 127].

To mention typical parameters for 1024-composites (see Chapter 5), we may expect $R \approx 10^{15}$, $U \approx 10^{10}$, and there are on the order of $B \approx 10^8$ instances of this problem (with different progressions and thresholds) to be solved. Thus, overall we need to check each of the 10^{23} sieve locations in order to find out which are divisible by about 10^9 possible prime numbers — hence the challenging computational magnitude of the problem.

1.6.2 Traditional sieving

The traditional method of performing the sieving task is, essentially, a variant of the algorithm for finding prime integers devised by Eratosthenes of Cyrene, 276–194 BC (see [20]). It proceeds as follows. An array of accumulators $C[a]$ is initialized to 0. Then, the progressions P_i are considered one by one, and for each P_i the indices $a \in P_i$ are calculated and the value $\log_c p_i$ is added to every such $C[a]$. Finally, the array is scanned to find the a values where $C[a] > T$. When looking at a specific P_i its members can be enumerated very efficiently, so the amortized cost of a $\log_c p_i$ contribution is low.

When this algorithm is implemented on a PC, we cannot apply it to the full range $a = 0, \ldots, R-1$ since there would not be enough RAM to store R accumulators (and if there was, the incessant cache misses would greatly degrade performance). Thus, the range is broken into smaller chunks, each of which is processed as above. However, if the chunk size is not much larger than U (i.e., the typical period of the progressions) then most progressions make very few contributions, if any, to each chunk; this increases the amortized cost per contribution. Thus, a huge amount of memory is required to implement this algorithm efficiently, both for the accumulators and for storing the list of progressions. As Bernstein [22] observed, this is inherently inefficient because each memory bit is accessed very infrequently.

Cost for 768-bit composites. Completing the sieving for 768-bit composites in 1 year using traditional sieving has been estimated to require 90,000 PC computers with 5GB of fast RAM each [130]. In prices circa 2003 and assuming a fivefold improvement in the relevant PC performance criteria since [130], this would cost about US$ 13M.[21]

Cost for 1024-bit composites. The cost of traditional sieving for 1024-bit composites is prohibitive, as shown in the following simple lower bound. In the algebraic (resp., rational) sieves, on average each sieve location gets a contribution from 7 (resp., 3) progressions with odd p_i. Suppose that each such contribution takes just 1ns on average to process (in practice it take significantly longer, due to the chunking described above and the non-local memory access pattern). Then the total running time is $(6/\pi^2) \cdot R \cdot B \cdot (7+3) \cdot 1\text{ns} \approx 57$ million years, assuming the same parameters as TWIRL (see §5.5). To implement this on a commodity PC computers we would need 10GB of main memory just for storing the pairs representing the progressions, and additional DRAM for storing the accumulators, for a total cost of about US$ 2,000 per PC in today's prices. Thus, the cost of employing enough PCs in parallel to complete the sieving in

[21] A similar figure is obtained analogously to the next paragraph when using the 768-bit parameters suggested in Chapter 5 for line sieving instead of the extrapolated special-q sieving used in [130].

1 year, disregarding operational costs such as power, would be about $(57 \cdot 10^6) \cdot \$2{,}000 \approx \$1 \cdot 10^{11}$ with these parameters.[22] This lower bound is consistent with prediction by extrapolation [197], which yields an estimate of about US$\$\, 10^{12}$ in current terms.

1.6.3 Historical sieving devices

The sieving task used in number-theoretical algorithms (ranging from Eratosthenes's algorithm to finding prime to the Number Field Sieve) has a very regular structure, which can be exploited by automated mechanical or electronic means. During the last century, severalsuch devices and been devised and built. These first such devices were used for factorization using Fermat's method, where the goal of sieving is to identify squares in an arithmetic progression by testing quadratic residuosity modulo various small primes. Later, similar sieving tasks arose in newer factoring algorithms such as the Continued Fraction and the Quadratic Sieve. These devices all predate the Number Field Sieve, but in principle they could be applied to it as well. In the following we briefly survey various special-purpose sieving devices which are of historical interest. The "L/s" value, where given, is the throughput of the machine in term of tested sieve locations per second. For further details and resources, as well as a broader scope, see my on-line annotated taxonomy of special-purpose cryptanalytic devices [203].

Employing mechanical devices for the sieving task was apparently first proposed by Frederick William Lawrence in 1896 [116]. In 1912, following the publication of a French translation, three prototype devices were independently built: by André Gérardin (the translator), by Maurice Kraitchik, and by Pierre and Eugène Olivier Carissan (see [184]). These three devices were rough prototypes that relied on a human observer and suffered from mechanical difficulties.

Machine á Congruences. This device, built by Eugène Olivier Carissan in 1919 [42] as an improvement upon his brother's earlier design, is the first known sieving machine that operated successfully. It consists of 14 concentric brass rings, and employs electrical switches to identify events corresponding to (likely) squares in the Fermat method.[23]

Lehmer's electromechanical sieves. In the 1920's and 1930's, D. N. Lehmer and D. H. Lehmer built several mechanical sieving devices, employing various approaches: bicycle chains (1926 [117][118], 60 L/s), gears and photoelectric detectors (1932 [119][118][120], 6000 L/s) and movie films (1936). After implementing a sieve program on the ENIAC (1946 [123]), Standards Western Automatic Computer (SWAC) ([120], 1450 L/s), and the Illiac IV ([122]; $15 \cdot 10^6$ L/S), D. H. Lehmer also built electronic sieving devices using delay lines ("DLS-127", 1965 [121], 10^6 L/s), and partially built a device based on shift-register integrated circuits ("SRS-181", 1970's, $1.6 \cdot 10^7$ L/s predicted).[24]

[22] A different NFS parameter choice may somewhat reduce the cost, as may the use of special-q lattice sieving [129], but neither is expected to dramatic increase the feasibility.

[23] For details see the investigative report of H. C. Williams and Shallit [184].

[24] See also the reviews by Lehmer [123], Patterson [165] and Stephens [200].

Extended Precision Operand Computer, AKA Georgia Cracker. This is a 128-bit, parallel special-purpose computer for factoring using the Continued Fraction method, built by Jeffrey Smith and Samuel Wagstaff in 1982–3 [198][170].

A High Performance Factoring Machine. This is a 256-bit general-purpose computer designed by W. Rudd, D. A. Buell and D. M. Chiarull [178] to perform factorization 10 times faster than existing general-purpose computers, with a low construction cost.

Quasimodo ("quadratic sieve motor"). This device, designed by C. Pomerance, J. W. Smith and R. Tuler [169][168], was built but never functioned properly and was superceded by the arrival of more efficient general-purpose computers.

UMSU, OASiS and SSU. In 1981–2, H. C. Williams and C. D. Patterson [165, §4] constructed UMSU ("University of Manitoba Sieve Unit"), an extended variant of Lehmer's aborted SRS-181. It employed a hybrid approach, integrating dedicated shift-registers ICs with a general-purpose microcomputer to sieve at $1.33 \cdot 10^8$ L/s. This was further enhanced by H. C. Williams and A. Stephens, yielding OASiS ("Open Architecture Sieve System") [200], which reached $2.15 \cdot 10^8$ L/s. Subsequently, C. D. Patterson [165] designed and fabricated a VLSI sieving device with a rate of $4.22 \cdot 10^9$ L/s (for one full board), in which each each custom-made IC contained multiple shift registers and the associated control logic.

Notably, most of the aforementioned devices address somewhat different sieving problems than those that arise in NFS; in particular, they look for sieve locations that are members of some progression (p_i, r_i) for *every* p_i. That is, they compute set intersections, rather than the finer task of logarithm-summation and threshold-comparison needed for the Number Field Sieve. Also, the scales of the factor base (i.e., number and period of progressions) are much smaller than those of interest for present factorization challenges. The resulting architectures are thus very different from those we shall discuss in the following chapters, and many of the techniques introduced in those works are not applicable or do not scale to sizes we are interested in.

1.6.4 TWINKLE

The TWINKLE design, devised by Shamir [185] and improved and analyzed by Lenstra and Shamir [130], takes another approach to sieving by enlisting the power of modern VLSI and opto-electric technology. Each TWINKLE device consists of a wafer containing numerous independent cells, each in charge of a single progression P_i. After initialization the device operates synchronously for R clock cycles, corresponding to the sieving range $\{0 \le a < R\}$. At clock cycle a, the cell in charge of the progression P_i emits the value $\log p_i$ iff $a \in P_i$. The values emitted at each clock cycle are summed to obtain $\lambda(x)$, and if this sum exceeds the threshold T then the integer a is reported. This event is announced back to the cells, so that the i values of the pertaining P_i is also reported.

The global summation is done by analog electro-optical means: in order to "emit" the value $\log p_i$, a cell flashes an internal LED whose intensity (after appropriate filtering) is proportional to

$\log p_i$. A light sensor above the wafer measures the total light intensity in each clock cycle, which is proportional to $\lambda(a)$, and reports a success when this exceeds a given threshold T. The cells themselves are implemented by simple registers and ripple adders. To support the optoelectronic operations, it was originally proposed to implement the cells on Gallium Arsenide wafers (rather than standard silicon wafers). The physical structure and the details of the efficient cell designs, as well as various optimizations, are given in [185, 130]. Note that TWINKLE exchanges the roles of space and time compared to traditional sievers:

	Traditional	**TWINKLE**
Sieve locations	Space (accumulators)	Time
Progressions	Time	Space (cells)

Cost for 768-bit composites. The cost of TWINKLE for 768-bit composites was analyzed in [130]. It was estimated that to complete the factorization in 1 year, one can employ 2,500 TWINKLE devices (using an optimized variant of the design). However, these devices would need to be supported by auxiliary computation that, when performed using standard PCs, would require about 40,000 PCs with 5GB of memory each. In today's prices, and assuming a fivefold increase since [130] in the performance of all components, the total cost for completion in 1 year would be on the order of US$ 8M (excluding the non-recurring R&D cost).

Notes. Despite the novel and effective use of non-electronic physical phenomena, TWINKLE offers a relatively modest improvement over traditional sieving for large composites. TWINKLE relies on auxiliary computers for continuously preparing and reloading its input (especially for large composites, where there is not even room for enough cells to represent all the progressions), and these auxiliary computers turn out to form a bottleneck.

1.6.5 FPGA-based serial sieving

Kim and Mangione-Smith [104] described a sieving device employing off-the-shelf parts, namely Field Programmable Gate Array (FPGA) chips. The device uses classical sieving, without time-memory reversal. By employing several FPGA chips, each connected to multiple fast SRAM chips, it achieves a very high memory bandwidth and is subsequently claimed to be only 6 times slower than TWINKLE for 512-bit composites. As this implementation is heavily tied to a specific hardware platform, it is unclear how it scales to larger parallelism and larger sieving problems.

1.6.6 Mesh-based sieving

An alternative approach to highly parallel sieving hardware is the mesh-based sieving proposed by Geiselmann and Steinwandt [74], following a related proposal by Bernstein[25] [22], and subsequently improved in [76]. Here we only sketch the basic idea in its simplest form, and refer the reader to [74, 76] for further details and improvements.

[25] This is merely hinted in [22], via the phrase "sieving via Schimmler's algorithm".

The device consists of a two-dimensional mesh of $s \times s$ nodes, with each node connected only to its (at most) 4 neighbors.[26] The nodes implement a routing network that can carry packets from any node to any node by a series of hops between adjacent cells. The range of sieve locations $a \in \{0, \ldots, R-1\}$ is partitioned into segments of size s^2, and each segment is handled separately as follows.

The s^2 sieve locations in the current segment are assigned to the s^2 mesh modes by a bijective mapping. In addition, the mesh contains representations of all the progressions in the factor base, partitioned among the s^2 nodes and stored by some efficient means. Each node performs two functions. First, it scans the progressions represented within it, identifies the ones which contain some sieve location(s) inside the current segment, and for each such case emits a corresponding packet; the packet specifies the corresponding contribution $\log_c p_i$, and the address of the mesh node in charge of sieve location a. Each node contains an accumulator for the sieve location a assigned to it, and for each incoming packet addressed to this node, it adds the transmitted $\log_c p_i$ to its accumulator and discards the packet. Thus, once all the packets have been generated, routed and processed, the accumulators contain the $g(a)$ values and can be tested against the threshold. This process is repeated segment by segment.

The original sieving-based architecture proposed in [74] was evaluated only for 512-bit composites (in which case its cost is comparable to TWINKLE's). It suffers from grave scalability issues that appear to make it inapplicable to much larger composites. An improved architecture, incorporating ideas and techniques from the publications summarized in this dissertation, was subsequently published by Geiselmann and Steinwandt [76]; see §6.3.

1.6.7 Relation collection without sieving

The task of relation collection can also be carried out more directly, without the use of a sieve. For example, one can simply evaluate $N_{\mathbf{a}}(a, b)$ and $N_{\mathbf{r}}(a, b)$ and identify smooth values using factoring methods that are efficient for smooth integers, such as the Elliptic Curve Method. As has been pointed out by Bernstein [22], this essentially eliminates storage costs, and is thus asymptotically advantageous when considering throughput cost (see §1.4 and §5.2.1.4. However, it is widely believed (with no contrary evidence) that this approach is not competitive with sieving the for composite sizes of interest, i.e., those that are presently feasible to factor by *any* known algorithm.

1.7 The NFS linear algebra step

In the NFS linear algebra step, we are given a $D \times D$ matrix A over GF(2), whose columns correspond to cycles built from the relations found in the preceding sieving step. This matrix is large but sparse, with a non-uniform distribution of row densities. Our goal is to find a few vectors in the kernel of A, i.e., vectors $\vec{w}$ such that $A\vec{w} = \vec{0}$. Each such vector is the characteristic vector of a set of cycles fulfilling (1.1).

[26] Higher dimensional meshes, where available, would reduce the asymptotic cost.

Typical matrix sizes we consider for 1024-bit composites are $D \approx 10^7$ to $D \approx 10^{10}$, with column weights on the order of 100 (see Chapter 5). These huge magnitudes exceed typical scientific computing workloads, and raise several crucial considerations:

Scalability. One has to either devise an efficient distributed algorithm that can cope with low-bandwidth inter-device connections, or build very dense monolithic devices capable of storing and processing the large data sets involved.

Fault tolerance. A computation of this magnitude cannot be expected to complete without fault, unless suitable means for error recovery are employed in software or hardware.

Parallelizing and amortizing access to data. The cost of storing the input matrix is, by itself, very large compared to typical processing units (e.g., CPU). As observed by Bernstein [22], it is crucial to amortize this cost by having numerous processing units that access the same storage in parallel.

The best approaches known for solving very sparse linear systems over finite fields are the conjugate gradient and Lanczos methods (in particular, the block Lanczos algorithm [142, 49]), and the Wiedemann algorithm and its variants.[27] We shall focus on the latter, as the simpler structure of its core operations makes it more suitable for hardware implementation.

1.7.1 The block Wiedemann algorithm

The block Wiedemann algorithm is based on Wiedemann's original algorithm [216], generalized to a parallelizable ("blocked") variant by Coppersmith [48]. It was partially analyzed, extended and first implemented by Kaltofen et al. [98][99], and further extended and fully analyzed by Villard for the homogeneous [212][211] and non-homogeneous [213] cases.

The block Wiedemann algorithm is a randomized Las Vegas algorithm, i.e., the correctness of the answer is readily verified. Its basic form is as follows. We shall be working over a finite field $\mathrm{GF}(q)$, usually with $q = 2$. The input is a matrix $A \in \mathrm{GF}(q)^{D \times D}$. For appropriately chosen integers m, n, and $\delta \stackrel{\text{def}}{=} D/n + D/m + O(1)$:

1. Choose random matrices $U \in \mathrm{GF}(q)^{m \times D}$ and $V \in \mathrm{GF}(q)^{D \times n}$. Compute $\bar{V} = AV$.
2. Compute $H_i = UA^i\bar{V} \in \mathrm{GF}(q)^{m \times n}$ for $i = 0, \ldots, \delta - 1$.
3. Compute the generating vector polynomial of the sequence H_i, i.e.,

$$\vec{g}(\lambda) = \sum_{i=0}^{d} \vec{g}_i \lambda^i \in (\mathrm{GF}(q)[\lambda])^n$$

[27] See [113] for a unified analytic approach. Structured Gaussian elimination is used as often employed as a preprocessing stage — see [43].

with $d < D/n$ such that

$$\sum_{i=0}^{d} H_{i+j}\vec{g}_i = \sum_{i=0}^{d} UA^{i+j}\bar{V}\vec{g}_i = 0 \quad \text{for all } j \in \{0, \ldots, \delta - n\} \ . \tag{1.2}$$

With high probability the left-projection by U in (1.2) does not affect the minimal recurrence relation of the sequence, and then $\vec{g}(\lambda)$ is also a generating vector polynomial for the sequence $\{A^i\bar{V}\}_i$:

$$\sum_{i=0}^{d} A^{i+j}\bar{V}\vec{g}_i = 0 \quad \text{for all } j \in \{0, \ldots, \delta - n\} \ . \tag{1.3}$$

4. Let ℓ be the smallest non-negative integer such that $\vec{g}_\ell \neq 0$. Compute

$$\hat{\vec{w}} = \sum_{i=0}^{d-\ell} A^i V \vec{g}_{\ell+i} \ .$$

Note that as a special case of (1.3) for $j = 0$:

$$A^{\ell+1}\hat{\vec{w}} = \sum_{i=\ell}^{d} A^i AV\vec{g}_i = \sum_{i=\ell}^{d} A^i\bar{V}\vec{g}_i = \sum_{i=0}^{d} A^i\bar{V}\vec{g}_i = 0 \ .$$

Thus, if $\hat{\vec{w}} \neq 0$ (which indeed holds with high probability), for some $i \in \{0, \ldots, \ell\}$ and $\vec{w} = A^i\hat{\vec{w}}$, we have $\vec{w} \neq 0$ and $A\vec{w} = 0$.

One thus obtains, with high probability, a non-zero vector in the kernel of A. In the Number Field Sieve we usually need several such (linearly independent) vectors, since some of them will correspond to trivial relations among the elements of the factor base. To obtain up to n such vectors, steps 3 and 4 can be extended by use of matrix generating polynomial [211, §3.2]. In the general case this requires preconditioning [98][211][45].

The guarantee of high probability of success provably holds for sufficiently large m, n. Conversely, for the case $m = n = 1$, corresponding to Wiedemann's original algorithm [216], the algorithm needs to be partially iterated multiple times, roughly doubling the work.[28]

1.7.2 Complexity of the block Wiedemann algorithm

The algorithm treats the matrix A as a black box, and uses it only as an operator applied to the vector space $\mathrm{GF}(q)^D$ by multiplication. Accordingly, the algorithm is particularly efficient when multiplication of A by a vector can be performed quickly — as is indeed the case for the Number

[28] The latter is heuristic; a full analysis of this case is not known.

Field Sieve, due to the sparseness of A. Accordingly, we shall evaluate its complexity in terms of the total number of matrix-by-vector multiplications, plus the number of auxiliary operations over $GF(q)$.

Step 1 requires n matrix-by-vector multiplications, and choice of $(n+m)D$ random elements in $GF(q)$.

Step 2 can be performed by considering each column $\vec{v}_j$ of $\bar{V}$ separately. For each of $j = 1, \ldots, n$, and for each $i = 0, \ldots, \delta - 1$, compute $A^i \vec{v}_j$ through repeated multiplication by A, and take its inner product with each of the rows of U. Overall, this requires $\delta n = (1 + n/m)D + O(n)$ matrix-by-vector multiplications and $\delta m D = (1 + m/n)D^2 + O(nD)$ operations in GF(q). In practice, as a heuristic optimization the matrix U is often taken to be very sparse rather than fully random; this makes the inner products essentially free.[29]

In Coppersmith's original algorithm [48], step 3 is carried out by a heuristic generalization of the Berlekamp-Massey algorithm [135]. Through more advanced techniques [98][211, §3], step 3 can be provably performed at a cost of $O(\,(n+m)^2 D \log^2 D \log\log D\,)$ operations in GF(q) using a probabilistic algorithm, or $O(\,(n+m)^2 (m/n) D \log^2((m/n)D) \log\log D\,)$ operations using a deterministic algorithm.

Similarly to step 1, step 4 can be performed by considering each column $\vec{v}_j$ of V separately. For each of $j = 1, \ldots, n$, and for each $i = 0, \ldots, \delta - 1$, compute $A^i \vec{v}_j$ through repeated multiplication by A, multiply it by the scalar $(\vec{g}_i)_j$, and accumulate the result for each j. Overall, this requires $(d - \ell)n < D$ matrix-by-vector multiplications and $\delta(d-\ell)n(D+1) = D^2 + O(D)$ operations in GF(q). If multiple kernel elements are needed, only the latter need to be repeated; in the case of GF(2), our primary interest, they are trivial.

Overall, for reasonably small n, m (i.e., $n, m \ll \sqrt{D \log^2 D \log\log D}$) and nontrivial matrices, the cost of the block Wiedemann algorithm is dominated by the matrix-by-vector multiplications in steps 1 and 3. If moreover $m \gg n \gg 1$, then there are roughly $2D$ such multiplications, divided into $2n$ chains each of the form

$$A\vec{v}_i\,,\ A^2\vec{v}_i\,,\ A^3\vec{v}_i\,,\ \ldots,\ A^{\delta'}\vec{v}_i \quad \text{for } \delta' \approx D/n\ . \tag{1.4}$$

Note that we compute each such chain twice, once in step 1 and again in step 3, in order to avoid storing all the product vectors (see below).

For simplicity of calculation we shall subsequently always assume that the length δ' of every chain is exactly D/n. Also, for consistency with some of the published literature we shall denote $K \stackrel{\text{def}}{=} n$, termed the *blocking factor* of the algorithm.

The non-block case. In the case $m = n = 1$, corresponding to Wiedemann's original non-block algorithm, there are approximately $3D$ multiplications; moreover, the process may need to be repeated since there is a possibility of failure (when the generating polynomial of (1.2) is

[29] Partial justification for this practice is offered by the related variant of Villard [211, §10].

merely a factor of the minimal generating polynomial of (1.3); see [216]). Moreover, the Number Field Sieve requires several linearly independent kernel elements: only half of the vectors will yield a non-trivial congruence (see §1.5.3), and moreover certain NFS optimizations necessitate discarding most of the vectors. For example, in RSA-155 [44], a total of about 10 kernel vectors were needed. Fortunately, getting additional vectors is likely to be cheaper than getting the first one — roughly 1/3 the cost (this is implicit in [216, Algorithm 1], taking advantage of the aforementioned polynomial factors). As a crude approximation relying on the aforementioned RSA-155 figure, we may expect the number of multiplications to be roughly $3 \cdot \frac{10}{3} \cdot 2D = 20D$.

Space complexity. Apart from the matrix A, the algorithm stores only a handful of vectors and polynomials. Note that the matrix-by-vector products (1.4) are not stored; rather, we only store their inner products with some other vectors (step 2) or accumulate their scalar products (step 4), and discard the intermediate results.[30]

Thus, storage cost is dominated by the storage requirements of the input matrix A itself.

1.7.3 The reduced task

After the above reductions and simplifications, the main task of the linear algebra step in the Number Field Sieve can be stated simply as follows.

Given a matrix $A \in \mathrm{GF}(2)^{D \times D}$ and K vectors $\vec{v}_1, \ldots, \vec{v}_K \in \mathrm{GF}(2)^D$, for $K \gg 1$, do the following twice: perform K independent chains of D/K iterated matrix-by-vector multiplications, to compute the vectors

$$A\vec{v}_i\,,\ A^2\vec{v}_i\,,\ A^3\vec{v}_i\,,\ \ldots,\ A^{D/K}\vec{v}_i \quad \text{for } i = 1, \ldots, K$$

and apply some simple, local functions on-the-fly to these products.

When $K = 1$, in the context of NFS, there are roughly $20D$ iterated multiplications, in one fully sequential chain.

1.7.4 The traditional approach to the matrix step

A simple implementation of the matrix step on general-purpose computers puts a full copy of the matrix on a single computer, and executes the multiplication chains by the standard methods for sparse linear algebra on general-purpose computers (e.g., see Penninga [166] for details optimizations). Trivial parallelization is obtained by exploiting the blocking factor K to distribute the load to up to K separate processors. This approach is extremely inefficient, since each computer would hold enormous (subexponential) storage while accessing a single (constant-sized) word at time. Even by considering just the memory bandwidth, we get a lower bound on the throughput cost of this approach. For example, for 1024-bit composites and PCs circa 2002, we get a lower

[30] Indeed, had we stored the intermediate results, we could have avoided recomputing essentially the same products in both step 2 and step 4.

bound of $1.8 \cdot 10^5$ US\$ $\times$ years for the throughput-optimized matrix parameters of §5.4.1.2, and $3.7 \cdot 10^{11}$ US\$ $\times$ years for the runtime-optimized matrix.[31]

For large matrices, one can distribute the matrix storage among multiple interconnected computers, and perform a joint computation via a distributed algorithm. This technique was used by Lenstra et al. [126][24] for their implementation of NFS on a MasPar parallel computer, as well as in many of the subsequent factorization experiments and in other contexts (see Kaltofen and Lobo [99]). This approach scales more favorably than the trivial parallelization, both asymptotically and in practice. However, performance is limited by the bandwidth of the network links (see [35]), and most resources in the computation nodes are still severely underutilized.

1.7.5 Bernstein's mesh-based linear algebra circuit

In 2001, Bernstein [22] made the observation that if one considers throughput cost (see §1.4) then the cost of NFS can be reduced by appropriate parallelization. This takes advantage of the following. The two most expensive steps of NFS have high space complexity (subexponential in the size of the composite n), but most of this space is occupied by huge inputs that were computed in previous steps. In conventional architectures, only a few bits of storage are accessed by the processor at a given time, while the rest are just idly "twiddling their thumbs". If we could have numerous processing elements use a single copy of the input, the throughput cost would be reduced.

Addressing the linear algebra step, Bernstein proposed an architecture based on processing nodes connected via a mesh topology. Initially all the non-zero matrix entries, as well as a vector v, are loaded onto the mesh, one value per node. Then, Schimmler's mesh sorting algorithm [180] and local operations are used to move the transform these values in order to compute the product Av. We omit the details, as the routing-based architecture of Chapter 3 essentially supercedes this architecture.

Asymptotically, the architecture proposed in [22] provided an asymptotic improvement, whose magnitude depends on the method of comparison. In terms of the bit length of composites that can be factored at a given throughput cost, Bernstein claimed an asymptotic improvement factor of 3.01, whereas our alternative interpretation yields a more modest factor of 1.17 (see [131]).

Concretely, however, it turns out that reaping the benefits of this approach is not trivial and that a direct implementation is impractical, as shall be seen in §3.2.

[31] See our publication [131] for details; there, these matrices are termed "small" and "large" respectively.

Chapter 2

The TWIRL architecture for the NFS sieving step

מַרְאֵה הָאוֹפַנִּים וּמַעֲשֵׂיהֶם כְּעֵין תַּרְשִׁישׁ, וּדְמוּת אֶחָד לְאַרְבַּעְתָּן; [...] כִּי רוּחַ הַחַיָּה, בָּאוֹפַנִּים.
— יחזקאל א ט"ז–כ'

The appearance of the wheels and their work was like unto a beryl: and they four had one likeness; [...] for the spirit of the living creature was in the wheels. — *Ezekiel I 16–20 (ASV)*

2.1 Overview

In this chapter we describe a new hardware implementation of the NFS sieving step which, for pertinent problem sizes and technology, is 3-4 orders of magnitude more cost effective than the best previously published designs (such as the optoelectronic TWINKLE and the mesh-based sieving).

Based on a detailed analysis of all the critical components (but without an actual implementation), we believe that the NFS sieving step for 512-bit RSA keys can be completed in less than ten minutes by a $10K device. For 1024-bit RSA keys, analysis of the NFS parameters (backed by experimental data where possible) suggests that the sieving step can be completed in less than a year by a US$ 1.1M device.

This brings the NFS sieving step for 1024-bit composites to within practical reach. Combined with the results of subsequent chapters, we will be able to conclude that 1024-bit RSA keys can no longer be considered safe from well-funded adversaries.

Our approach is as follows. One lesson learned from Bernstein's mesh-based circuit for the matrix step [22] is that it is inefficient to have memory cells that are "simply sitting around, twiddling their thumbs" — if merely storing the input is expensive, we should utilize it efficiently by appropriate parallelization. The proposed new device, termed TWIRL[1], combines this intuition

[1] TWIRL stands for "The Weizmann Institute Relation Locator"

with the TWINKLE-like approach of exchanging time and space. Whereas TWINKLE tests sieve location one by one serially, the new device handles thousands of sieve locations in parallel at every clock cycle. The main difficulty is how to use a single copy of the input to solve many subproblems in parallel, without collisions or long propagation delays and while maintaining storage efficiency. We address this via a heterogeneous design that uses a variety of storage and routing circuits and takes advantage of available technological tradeoffs.

The resulting cost estimates show that the device is not only faster, but also smaller and easier to construct: for example, for 512-bit composites we can fit approximately 80 independent sieving devices on a 30cm single silicon wafer, whereas each TWINKLE device requires a full GaAs wafer. For 1024-bit composites, sieving can be performed at a surprisingly feasible cost of a few million $US, compared to the trillions previously posited.

2.2 Basic architecture

2.2.1 Approach

We begin by reviewing TWIRL's basic architecture, postponing the finer details to subsequent sections. The final design will be slightly different for the rational vs. algebraic sieves (see §1.5.3); we start by describing the rational sieve, and the alterations for the algebraic side are described in §2.3.5.

For the sake of concreteness we provide numerical examples for a plausible choice of parameters for 1024-bit composites. This choice will be discussed in §2.4; it is not claimed to be optimal, and all costs should be taken as rough estimates. The concrete figures will be enclosed in double angular brackets: $\langle\!\langle x \rangle\!\rangle_{\mathbf{r}}$ and $\langle\!\langle x \rangle\!\rangle_{\mathbf{a}}$ indicate a value x which is applicable to the algebraic and rational sieves respectively, and $\langle\!\langle x \rangle\!\rangle$ is applicable to both.

Recalling §1.6, we wish to solve $B \langle\!\langle \approx 2.7 \cdot 10^8 \rangle\!\rangle$ pairs of instances of the sieving problem, each of which has sieving line width $R \langle\!\langle = 1.1 \cdot 10^{15} \rangle\!\rangle$ and smoothness bound $U \langle\!\langle = 3.5 \cdot 10^9 \rangle\!\rangle_{\mathbf{r}} \langle\!\langle = 2.6 \cdot 10^{10} \rangle\!\rangle_{\mathbf{a}}$. For each instance, as input we are given arithmetic progressions P_i represented by pairs (p_i, r_i), and a threshold T.

Consider first a "pipeline-of-adders TWINKLE" device that handles one sieve location per clock cycle, like TWINKLE (see §1.6.4), but does so using a pipelined systolic chain of electronic adders.[2] Such a device would consist of a long unidirectional bus, $\lg T \langle\!\langle = 10 \rangle\!\rangle$ bits wide, that connects millions of conditional adders in series. Each conditional adder is in charge of one progression P_i; when activated by an associated timer, it adds the value[3] $\lfloor \log p_i \rceil$ to the bus. At time t, the z-th adder handles sieve location $t - z$. The first value to appear at the end of the pipeline is $\lambda(0)$, followed by $\lambda(1), \ldots, \lambda(R)$, one per clock cycle. See Fig. 2.1(a).

[2] This variant of TWINKLE was considered in [130], but deemed inferior in that context.

[3] $\lfloor \log p_i \rceil$ denote the value $\log_c p_i$ for some fixed c, rounded to the nearest integer.

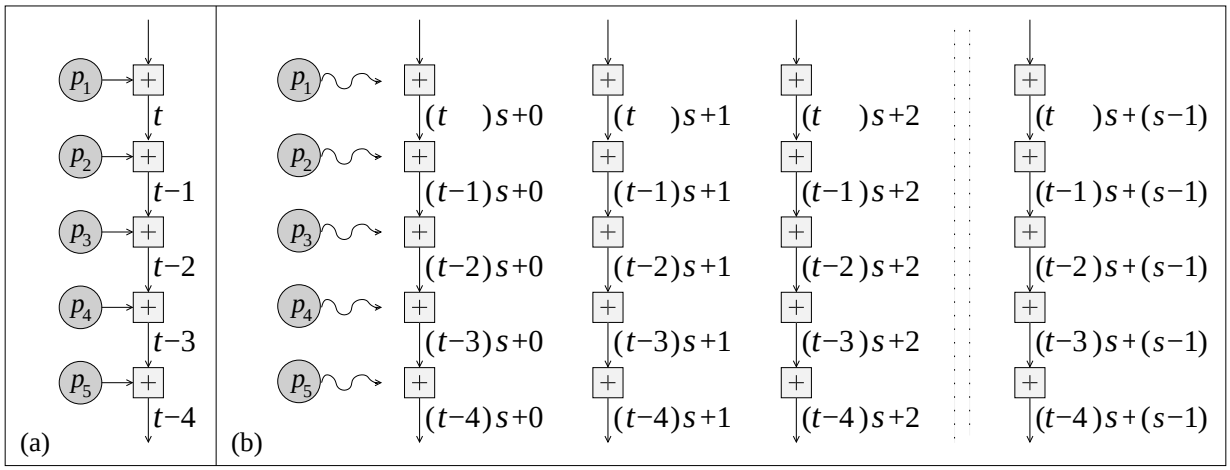

Figure 2.1: Flow of sieve locations through the device in (a) a chain of adders and (b) TWIRL.

We reduce the run time by a factor of $s \langle\!\langle = 4{,}096 \rangle\!\rangle_{\mathbf{r}} \langle\!\langle = 32{,}768 \rangle\!\rangle_{\mathbf{a}}$ by handling the sieving range $\{0, \ldots, R-1\}$ in chunks of length s, as follows. The bus is thickened by a factor of s to contain s logical lines of $\lg T$ bits each. As a first approximation (which will be altered later), we may think of it as follows: at time t, the z-th stage of the pipeline handles the sieve locations $(t-z)s+i$, $i \in \{0, \ldots, s-1\}$. The first values to appear at the end of the pipeline are $\{\lambda(0), \ldots, \lambda(s-1)\}$; they appear simultaneously, followed by successive disjoint groups of size s, one group per clock cycle. See Fig. 2.1(b).

Two main difficulties arise: the hardware has to work s times harder since time is compressed by a factor of s, and the additions of $\lfloor \log p_i \rceil$ corresponding to the same given progression P_i can occur at different lines of a thick pipeline. Our goal is to achieve this parallelism without simply duplicating all the counters and adders s times (which would keep the throughput cost the same). We thus replace the simple TWINKLE-like cells by other units which we call *stations*. Each station handles a small portion of the progressions, and its interface consists of bus input, bus output, clock and some circuitry for loading the inputs. The stations are connected serially in a pipeline, and at the end of the bus (i.e., at the output of the last station) we place a threshold check unit that produces the device output.

An important observation is that the progressions have periods p_i in a very large range of sizes, and different sizes involve very different design tradeoffs. We thus partition the progressions into three classes according to the size of their p_i values, and use a different station design for each class. In order of decreasing p_i value, the classes will be called *largish*, *smallish* and *tiny*.[4]

This heterogeneous approach leads to reasonable device sizes even for 1024-bit composites, despite the high parallelism: using standard CMOS VLSI technology, we can fit $\langle\!\langle 4 \rangle\!\rangle_{\mathbf{r}}$ rational-side TWIRL devices into a single 30cm silicon wafer (whose manufacturing cost is about $5,000 in

[4]These are not to be confused with the "large primes" of the high-level NFS algorithm. All the primes with which we are concerned here are below the smoothness bounds $U_{\mathbf{r}}$ or $U_{\mathbf{a}}$ respectively (see §1.5).

high volumes). Algebraic-side TWIRL devices use higher parallelism, and we fit only $\langle\!\langle 1 \rangle\!\rangle_{\mathbf{a}}$ of them into each wafer.

The following sections describe the hardware used for each class of progressions. The preliminary cost estimates that appear later are based on a careful analysis of all the critical components of the design, but due to space limitations we omit the descriptions of many finer details. Some additional issues are discussed in §2.3.

2.2.2 Largish primes

Progressions whose p_i values are much larger than s emit $\lfloor \log p_i \rceil$ values very seldom. For these largish primes, defined as $\langle\!\langle p_i > 5.2 \cdot 10^5 \rangle\!\rangle_{\mathbf{r}}$ $\langle\!\langle p_i > 4.2 \cdot 10^6 \rangle\!\rangle_{\mathbf{a}}$, it is beneficial to use expensive logic circuitry that handles many progressions but allows very compact storage of each progression. A sequence of circuit elements processes these progression, computes and schedules the corresponding contribution events, and adds them to the global bus passing through the station.

This station architecture is shown in Fig. 2.2. Each progression is represented as a *progression triplet* (to be defined later) that is stored in a memory bank, using compact DRAM[5] storage. The progression triplets are periodically inspected and updated by special-purpose processors, which identify emissions that should occur in the "near future" and create corresponding *emission triplets*. The emission triplets, in turn, are passed into *buffers* that merge the outputs of several processors, perform fine-tuning of the timing and create *delivery pairs*. Lastly, the delivery pairs are passed to pipelined *delivery lines*, consisting of a chain of *delivery cells* which carry the delivery pairs to the appropriate bus line and add their $\lfloor \log p_i \rceil$ contribution.

2.2.2.1 Scanning the progressions

The progressions are partitioned into many $\langle\!\langle 8{,}500 \rangle\!\rangle_{\mathbf{r}}$ $\langle\!\langle 60{,}000 \rangle\!\rangle_{\mathbf{a}}$ DRAM banks, where each bank contains some d progressions $\langle\!\langle 32 \le d < 2.2 \cdot 10^5 \rangle\!\rangle_{\mathbf{r}}$ $\langle\!\langle 32 \le d < 2.0 \cdot 10^5 \rangle\!\rangle_{\mathbf{a}}$. A progression P_i is represented by a progression triplet of the form (p_i, ℓ_i, τ_i), where ℓ_i and τ_i characterize the next element $a_i \in P_i$ to be emitted (which is not stored explicitly) as follows. The value $\tau_i = \lfloor a_i/s \rfloor$ is the time when the next emission should be added to the bus, and $\ell_i = a_i \bmod s$ is the number of the corresponding bus line. A processor repeats the following operations,[6] in a pipelined manner:

[5] DRAM ("Dynamic Random Access Memory") stores each bit as a charge level on a capacitor. It is the densest practical storage method available for electronic circuits, but is characterized by a high latency (often over 100 clock cycles).

[6] Additional logic related to reporting the sets $\{i : a \in P_i\}$, needed for the NFS linear algebra step, is described in §2.3.7.

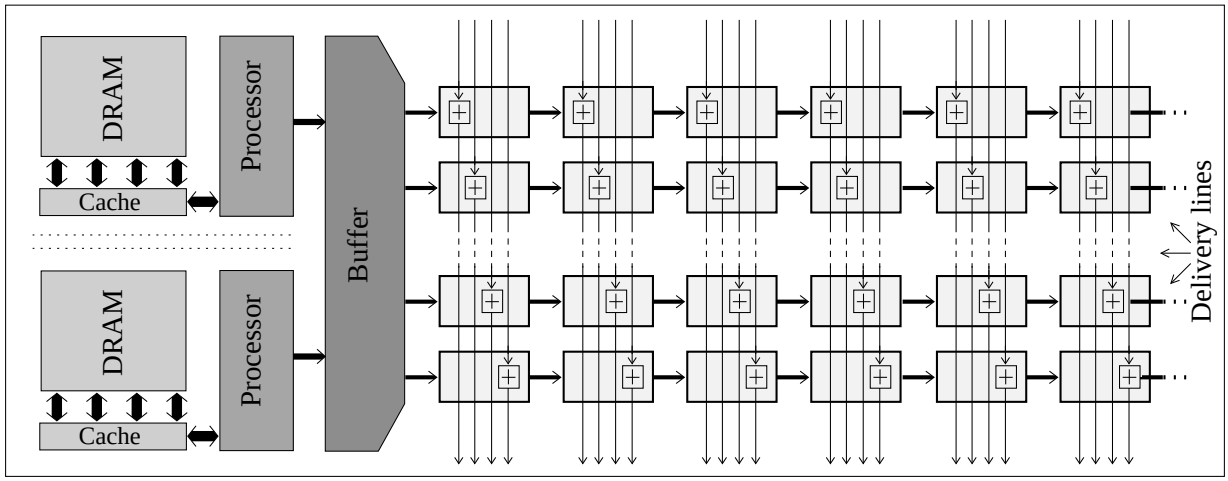

Figure 2.2: Schematic structure of a largish station

1. Read and erase the next state triplet (p_i, ℓ_i, τ_i) from memory.
2. Send an emission triplet $(\lfloor \log p_i \rceil, \ell_i, \tau_i)$ to a buffer connected to the processor.
3. Compute $\ell' \leftarrow (\ell + p) \bmod s$ and $\tau_i' \leftarrow \tau_i + \lfloor p/s \rfloor + w$, where $w = 1$ if $\ell' < \ell$ and $w = 0$ otherwise.
4. Write the triplet (p_i, ℓ_i', τ_i') to memory, according to τ_i' (see §2.2.2.2 below).

We wish the emission triplet $(\lfloor \log p_i \rceil, \ell_i, \tau_i)$ to be created slightly before time τ_i (earlier creation would overload the buffers, while later creation would prevent this emission from being delivered on time). Thus, we need the processor to always read from memory some progression triplet that has an imminent emission. For large d, the simple approach of assigning each emission triplet to a fixed memory address and scanning the memory cyclically would be ineffective. It would be ideal to place the progression triplets in a priority queue indexed by τ_i, but it is not clear how to do so efficiently in a standard DRAM due to its passive nature and high latency. However, by taking advantage of the unique properties of the sieving problem we can get a good approximation, as follows.

2.2.2.2 Progression storage

The DRAM bank is partitioned into *memory slots*, each of which can be empty (designated by a special value) or contain a progression triplet. The processor reads progression triplets from these memory slots in sequential cyclic order and at a constant rate ⟪of one slot every 2 clock cycles⟫. When the processor reads a progression from a non-empty slot, it updates the progression state as above and stores it at a different memory slot — namely, one that will be read slightly before

time τ_i'. In this way, after a short stabilization period the processor always reads triplets with imminent emissions. In order to have (with high probability) a free memory slot within a short distance of any slot, we increase the amount of memory ⟪by a factor of 2⟫; the progression is stored at the first unoccupied slot, starting at the one that will be read at time τ_i' and going backwards cyclically.

If there is no empty slot within ⟪64⟫ slots from the optimal designated address, the progression triplet is stored at an arbitrary slot (or a dedicated overflow region) and restored to its proper place at a later opportunity. When this happens we may miss a few emissions (depending on the implementation) — but not the whole progression. This happens very seldom,[7] and it is permissible to miss a few candidates.

Autonomous circuitry inside the memory routes the progression triplet to the first unoccupied position preceding the optimal one. To implement this efficiently we use a two-level memory hierarchy which is rendered possibly by the following observation. Consider a largish processor which is in charge of a set of d adjacent primes $\{p_{\min}, \ldots, p_{\max}\}$. We set the size of the associated memory to $p_{\max}/s$ triplet-sized words, so that triplets with $p_i = p_{\max}$ are stored right before the currently read slot; triplets with smaller p_i are stored further back, in cyclic order. By the density of primes, $p_{\max} - p_{\min} \approx d \cdot \ln(p_{\max})$. Thus triplet values are always stored at an address that precedes the current read address by at most $d \cdot \ln(p_{\max})/s$, or slightly more due to congestions. Since $\ln(p_{\max}) \le \ln(U)$ is much smaller than s, memory access always occurs at a small window that slides at a constant rate of one memory slot every ⟪2⟫ clock cycles. We may view the ⟪8,500⟫$_{\mathbf{r}}$ ⟪60,000⟫$_{\mathbf{a}}$ memory banks as closed rings of various sizes, with an active window "twirling" around each ring at a constant linear velocity.

Each sliding window is handled by a fast cache based on SRAM-type memory.[8] Occasionally, the window is shifted by writing the oldest cache block to DRAM and reading the next block from DRAM into the cache. Using an appropriate interface between the SRAM and DRAM banks (namely, read/write of full rows), this hides the high DRAM latency and achieves very high memory bandwidth. Also, this allows simpler and thus smaller DRAM.[9] Note that cache misses cannot occur. The only interface between the processor and memory are the operations "read next memory slot" and "write triplet to first unoccupied memory slot before the given address". The logic for the latter is implemented within the cache, using auxiliary per-triplet occupancy

[7] For instance, in simulations for primes close to ⟪20,000s⟫$_{\mathbf{r}}$, the distance between the first unoccupied slot and the ideal slot was smaller than ⟪64⟫$_{\mathbf{r}}$ for all but ⟪$5 \cdot 10^{-6}$⟫$_{\mathbf{r}}$ of the iterations. The probability of a random integer $x \in \{1, \ldots, x\}$ having k factors is about $(\log\log x)^{k-1}/(k-1)! \log x$. Since we are (implicitly) sieving over values of size about $x \approx$ ⟪10^{64}⟫$_{\mathbf{r}}$⟪10^{101}⟫$_{\mathbf{a}}$ which are "good" (i.e., semi-smooth) with probability $p \approx$ ⟪$6.8 \cdot 10^{-5}$⟫$_{\mathbf{r}}$ ⟪$4.4 \cdot 10^{-9}$⟫$_{\mathbf{a}}$, less than $10^{-15}/p$ of the good a's have more than 35 factors; the probability of missing other good a's is negligible.

[8] SRAM ("Static Random Access Memory") stores each bit using (typically) 6 transistors. It is characterized by low latency but larger area per bit compared to DRAM (see Footnote 5).

[9] Most of the peripheral DRAM circuitry (including the refresh circuitry and column decoders) can be eliminated, and the row decoders can be replaced by smaller stateful circuitry. Thus, the DRAM bank can be smaller than standard designs. For the stations that handle the smaller primes in the "largish" range, we may increase the cache size to d and eliminate the DRAM.

flags and local pipelined circuitry.[10]

2.2.2.3 Buffers

A buffer unit receives emission triplets from several processors in parallel, and sends delivery pairs to several delivery lines. Its task is to convert emission triplets into delivery pairs by merging them where appropriate, fine-tuning their timing and distributing them across the delivery lines: for each received emission triplet of the form $(\lfloor \log p_i \rceil, \ell, \tau)$, the delivery pair $(\lfloor \log p_i \rceil, \ell)$ should be sent to some delivery line (depending on ℓ) at time exactly τ.

Buffer units can be realized as follows. All incoming emission triplets are placed in a parallelized priority queue indexed by τ, realized as a small mesh whose rows are continuously bubble-sorted and whose columns undergo random local shuffles. Entries are injected at the top of the mesh, and elements with small τ values "fall" to the bottom rows. The elements in the bottom rows are tested for τ matching the current time, and the matching ones are passed to a pipelined network that sorts them by ℓ, merges where needed and passes them to the appropriate delivery lines. Due to congestions some emissions may be late and thus discarded; since the inputs are essentially random, with suitable parameters this is a rare event.

The size of the buffer depends on the typical number of time steps that an emission triplet is held until its release time τ (which is fairly small due to the design of the processors), and on the rate at which processors produce emission triplets ⟪about once per 4 clock cycles⟫.

2.2.2.4 Delivery lines

A delivery line receives delivery pairs of the form $(\lfloor \log p_i \rceil, \ell)$ and adds each such pair to bus line ℓ exactly $\lfloor \ell/k \rfloor$ clock cycles after its receipt. It is implemented as a one-dimensional array of cells placed across the bus, where each cell is capable of containing one delivery pair. Here, the j-th cell compares the ℓ value of its delivery pair (if any) to the constant j. In case of equality, it adds $\lfloor \log p_i \rceil$ to the bus line and discards the pair. Otherwise, it passes it to the next cell, as in a shift register.

Overall, there are $\langle\!\langle 2{,}100 \rangle\!\rangle_{\mathbf{r}}$ $\langle\!\langle 15{,}000 \rangle\!\rangle_{\mathbf{a}}$ delivery lines in the largish stations, and they occupy a significant portion of the device. §2.3.1 describes the use of interleaved carry-save adders to reduce their cost, and §2.3.5 nearly eliminates them from the algebraic sieve.

2.2.2.5 Notes

In the description of the processors, DRAM and buffers, we took the τ values to be arbitrarily large integers designating clock cycles. Actually, it suffices to maintain these values modulo some

[10] It needs to inspect just ⟪64⟫ slots, and this is easily done, e.g., using a search tree (of logarithmic depth) for addressing, and a broadcast channel for data.

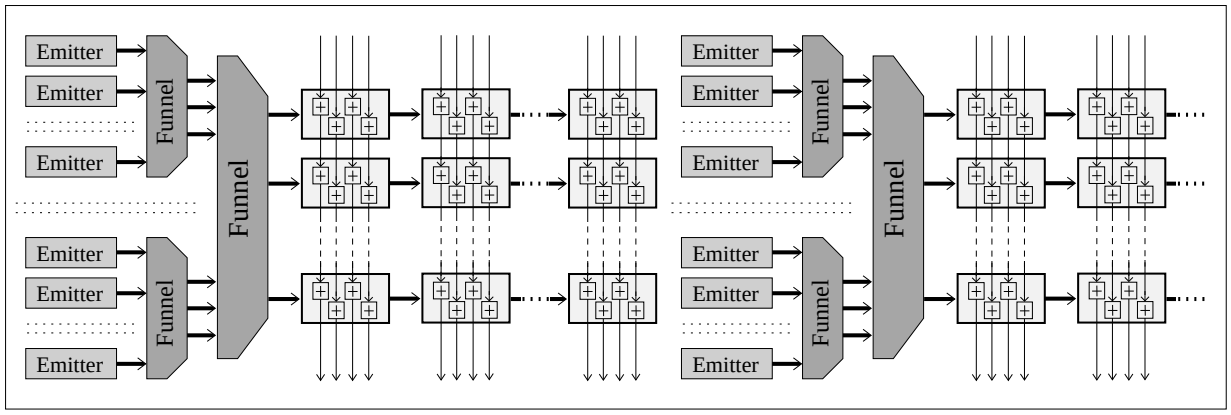

Figure 2.3: Schematic structure of a smallish station

integer $\langle\!\langle 2048 \rangle\!\rangle$ that upper bounds the number of clock cycles from the time a progression triplet is read from memory to the time when it is evicted from the buffer. Thus, a progression occupies only $\lg p_i + \langle\!\langle \lg 2048 \rangle\!\rangle$ DRAM bits for the triplet, plus $\lg p_i$ bits for re-initialization (see §2.3.4).

The amortized circuit area per largish progression is $\Theta(s^2(\log s)/p_i + \log s + \log p_i)$.[11] For fixed s this equals $\Theta(1/p_i + \log p_i)$, and indeed for large composites the overwhelming majority of progressions $\langle\!\langle 99.97\% \rangle\!\rangle_{\mathbf{r}}$ $\langle\!\langle 99.98\% \rangle\!\rangle_{\mathbf{a}}$ will be handled in this manner.

2.2.3 Smallish primes

For progressions with p_i close to s ($\langle\!\langle 256 < p_i < 5.2 \cdot 10^5 \rangle\!\rangle_{\mathbf{r}}$ $\langle\!\langle 256 < p_i < 4.2 \cdot 10^6 \rangle\!\rangle_{\mathbf{a}}$), each processor can handle very few progressions because it can produce at most one emission triplet every $\langle\!\langle 2 \rangle\!\rangle$ clock cycles. Thus, the amortized cost of the processor, memory control circuitry and buffers is very high. Moreover, each such progression causes emissions so often that communicating these emissions to distant bus lines (which is necessary if the state of each progression is maintained at some single physical location) would involve enormous communication bandwidth. We thus introduce a different station design to deal with the smallish primes (see Fig 2.3). The changes, compared to the largish stations, are as follows.

2.2.3.1 Emitters and funnels

The first change is to replace the combination of the processors, memory and buffers by other units. Delivery pairs are now created directly by *emitters*, which are small circuits that handle a single progression each (as in TWINKLE). An emitter maintains the state of the progression using internal registers, and periodically emits delivery pairs of the form $(\lfloor \log p_i \rceil, \ell)$ which indicate

[11] The frequency of emissions is s/p_i, and each emission occupies some delivery cell for $\Theta(s)$ clock cycles. The last two terms are due to DRAM storage, and have very small constants.

that the value $\lfloor \log p_i \rceil$ should be added to the ℓ-th bus line some fixed time interval later. §2.3.2 describes a compact emitters design.

Each emitter is continuously updating its internal counters, but it creates a delivery pair only once per roughly $\sqrt{p_i}$ (i.e., between $\langle\!\langle 8 \rangle\!\rangle_{\mathbf{r}}$ and $\langle\!\langle 512 \rangle\!\rangle_{\mathbf{r}}$ clock cycles; but see §2.2.3.2 fur further adjustment). It would be wasteful to connect each emitter to a dedicated delivery line. This is solved using *funnels*, which "compress" their sparse inputs as follows. A funnel has a large number of input lines, connected to the outputs of many adjacent emitters; we may think of it as receiving a sequence of one-dimensional arrays, most of whose elements are empty. The funnel outputs a sequence of much shorter arrays, whose non-empty elements are exactly the non-empty elements of the input array received a fixed number of clock cycle earlier. The funnel outputs are connected to the delivery lines. §2.3.3 describes an implementation of funnels using modified shift registers.

2.2.3.2 Duplication

The other major change is a duplication of the progression states, in order to move the sources of the delivery pairs closer to their destination and reduce the cross-bus communication bandwidth. Each progression P_i is handled by $n_i \approx s/\sqrt{p_i}$ independent emitters[12] which are placed at regular intervals across the bus. Accordingly we fragment the delivery lines into segments that span $s/n_i \approx \sqrt{p_i}$ bus lines each. Each emitter is connected (via a funnel) to a different segment, and sends emissions to this segment every $p_i/sn_i \approx \sqrt{p}$ clock cycles. As emissions reach their destination quicker, we can decrease the total number of delivery lines. Also, the reduction of each emitter's emission frequency by a factor of n_i allows us to handle p_i close to (or even smaller than) s. Overall there are $\langle\!\langle 501 \rangle\!\rangle_{\mathbf{r}}$ delivery lines in the smallish stations, broken into segments of various sizes.

2.2.3.3 Notes

Asymptotically the amortized circuit area per smallish progression is $\Theta(\,(s/\sqrt{p_i} + 1)\,(\log s + \log p_i)\,)$.

The term 1 is less innocuous than it appears — it hides a large constant (roughly the size of an emitter plus the amortized funnel size), which dominates the cost for sufficiently large p_i.

2.2.4 Tiny primes

For very small primes, the amortized cost of the duplicated emitters, and in particular the related funnels, becomes too high. On the other hand, such progressions cause several emissions at every clock cycle, so it is less important to amortize the cost of delivery lines over several progressions.

[12] $\langle\!\langle n_i = s/2\sqrt{p_i} \rangle\!\rangle$ rounded to a power of 2 (see §2.3.2), which is in the range $\langle\!\langle \{2, \ldots, 128\} \rangle\!\rangle_{\mathbf{r}}$.

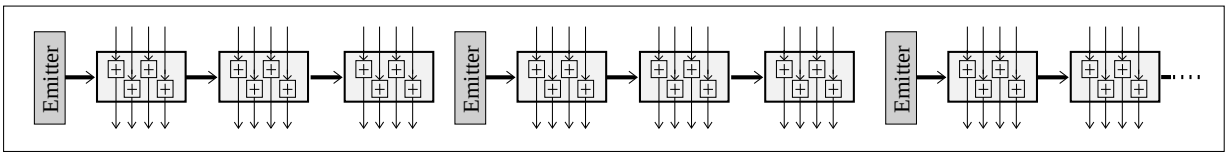

Figure 2.4: Schematic structure of a tiny station, for a single progression

This leads to a third station design for the tiny primes $\langle\!\langle p_i < 256 \rangle\!\rangle$. While there are very few such progressions and each corresponding $\lfloor \log p_i \rceil$ contribution to the sieve locations is small, the total contribution of such primes is significant since these contribution events happen very frequently.

Each tiny progression is handled independently, using a dedicated delivery line. The delivery line is partitioned into segments of size somewhat smaller than p_i,[13] and an emitter is placed at the input of each segment, without an intermediate funnel (see Fig 2.4). These emitters are a degenerate form of the ones used for smallish progressions (see §2.3.2). Here we cannot interleave the adders in delivery cells as done in largish and smallish stations, but the carry-save adders are smaller since they only (conditionally) add the small constant $\lfloor \log p_i \rceil$. Since the area occupied by each progression is dominated by the delivery lines, it is $\Theta(s)$ regardless of p_i.

2.3 Additional design considerations

Having presented the basic architecture, we proceed to provide additional details and enhancements.

2.3.1 Delivery lines

The delivery lines are used by all station types to carry delivery pairs from their source (buffer, funnel or emitter) to their destination bus line. Their basic structure is described in §2.2.2. We now describe methods for implementing them efficiently.

2.3.1.1 Interleaving

Most of the time the cells in a delivery line act as shift registers, and their adders are unused. Thus, we can reduce the cost of adders and registers by dropping most registers and interleaving the remaining, as follows. We use larger delivery cells that span $r \langle\!\langle = 4 \rangle\!\rangle_{\mathbf{r}}$ adjacent bus lines,[14] and contain an adder just for the q-th line among these, with q fixed throughout the delivery line and incremented cyclically in the subsequent delivery lines (see Figure 2.2). As a bonus, we now put every r adjacent delivery lines in a single bus pipeline stage, so that it contains one adder

[13] The segment length is the largest power of 2 smaller than p_i (see §2.3.2).
[14] This applies only to the rational sieve, due to cascading (see §2.3.5).

per bus line. This reduces the number of bus pipelining registers by a factor of r throughout the largish stations.

Since the emission pairs traverse the delivery lines at a rate of r lines per clock cycle, we need to skew the space-time assignment of sieve locations so that as distance from the buffer to the bus line increases, the "age" $\lfloor a/s \rfloor$ of the sieve locations decreases. More explicitly: at time t, sieve location a is handled by the $\lfloor (a \bmod s)/r \rfloor$-th cell[15] of one of the r delivery lines at stage $t - \lfloor a/sr \rfloor - \lfloor (a \bmod s)/r \rfloor$ of the bus pipeline, if it exists.

In the largish stations, the buffer is entrusted with the role of sending delivery pairs to delivery lines that have an adder at the appropriate bus line. An improvement by a factor of 2 is achieved by placing the buffers at the middle of the bus, with the two halves of each delivery line directed outwards from the buffer. In the smallish and tiny stations we do not interleave the adders: the travel distance of delivery pairs is already lower due to emitter duplication, and we wish to avoid the overhead of directing delivery pairs to an appropriate delivery line.[16]

Note that whenever we place pipelining registers on the bus, we must delay all downstream delivery lines connected to this buffer by a clock cycle. This can be done by adding pipeline stages at the beginning of these delivery lines.

2.3.1.2 Carry-save adders

Logically, each bus line carries a $\lg T \langle\!\langle = 10 \rangle\!\rangle$-bit integer. These are encoded by a redundant representation, as a pair of $\lg T$-bit integers whose sum equals the sum of the $\lfloor \log p_i \rceil$ contributions so far. The additions at the delivery cells are done using carry-save adders, which have inputs a, b, c and whose output is a representation of the sum of their inputs in the form of a pair e, f such that $e + f = a + b + c$. Carry-save adders are very compact and support a high clock rate, since they do not propagate carries across more than one bit position. Their main disadvantage is that it is inconvenient to perform other operations directly on the redundant representation, but in our application we only need to perform a long sequence of additions followed by a single comparison at the end. The number of bus wires is doubled, but these can be relegated to an extra layer of metal conductors in the VLSI process; we expect the bottleneck to remain at the logic/DRAM layer.[17]

To prevent wrap-around due to overflow when the sum of contributions is much larger than T, we slightly alter the carry-save adders by making their most significant bits "sticky": once the MSBs of both values in the redundant representation become 1 (in which case the sum is at least T), further additions do not switch them back to 0.

[15] This is changed in §2.3.2.2.

[16] Still, the number of adders can be reduced by attaching a single adder to several bus lines using multiplexers. This may impact the clock rate.

[17] Should this prove problematic, we can use the standard integer representation with carry-lookahead adders, at some cost in circuit area and clock rate.

2.3.2 Implementation of emitters

The designs of smallish and tiny progressions (see §2.2.3, 2.2.4) included *emitter* elements. An emitter handles a single progression P_i, and its role is to emit the delivery pairs $(\lfloor \log p_i \rceil, \ell)$ addressed to a certain group G of adjacent lines, $\ell \in G$. This section describes our proposed emitter implementation. For context, we first describe some less efficient designs.

2.3.2.1 Straightforward implementations

One simple implementation would be to keep a $\lceil \lg p_i \rceil$-bit register and increment it by s modulo p_i every clock cycle. Whenever a wrap-around occurs (i.e., this progression causes an emission), compute ℓ and check if $\ell \in G$. Since the register must be updated within one clock cycle, this requires an expensive carry-lookahead adder. Moreover, if s and $|G|$ are chosen arbitrarily then calculating ℓ and testing whether $\ell \in G$ may also be expensive. Choosing s, $|G|$ as power of 2 reduces the costs somewhat.

A different approach would be to keep a counter that counts down the time to the next emission, as in [185], and another register that keeps track of ℓ. This has two variants. If the countdown is to the next emission of this triplet regardless of its destination bus line, then these events would occur very often and again require low-latency circuitry (also, this cannot handle $p_i < s$). If the countdown is to the next emission into G, we encounter the following problem: for any set G of bus lines corresponding to adjacent residues modulo s, the intervals at which P_i has emissions into G are irregular, and would require expensive circuitry to compute.

2.3.2.2 Line address bit reversal

To solve the last problem described above and use the second countdown-based approach, we note the following: the assignment of sieve locations to bus lines (within a clock cycle) can be done arbitrarily, but the partition of wires into groups G should be done according to physical proximity. Thus, we use the following trick. Choose $s = 2^\alpha$ and $|G| = 2^{\beta_i} \approx \sqrt{p_i}$, for some integers $\alpha \langle\!\langle = 12 \rangle\!\rangle_{\mathbf{r}} \langle\!\langle = 15 \rangle\!\rangle_{\mathbf{a}}$ and β_i (where β_i depends on the progression P_i). The residues modulo s are assigned to bus lines with bit-reversed indices: sieve locations congruent to w modulo s are handled by the bus line at physical location $\mathrm{rev}(w)$, where $\mathrm{rev}(\cdot)$ denotes the bit-reversal of an a-bit string.[18]

Consequently, the j-th emitter of the progression P_i, $j \in \{0, \ldots, 2^{\alpha-\beta_i}\}$, is in charge of the j-th group of 2^{β_i} bus lines, i.e., it services the following sieve locations

$$G_{i,j} \stackrel{\text{def}}{=} \left\{ a \ : \ j \cdot 2^{\beta_i} \leq \mathrm{rev}(a \bmod s) < (j+1) \cdot 2^{\beta_i} \right\} \ .$$

The advantage of this choice is the following. Let $\Delta_i \stackrel{\text{def}}{=} \lfloor 2^{-\beta_i} p_i \rfloor$. Then:

[18] Let $w = \sum_{i=0}^{\alpha-1} c_i 2^i$ for some $c_0, \ldots, c_{\alpha-1} \in \{0,1\}$. Then $\mathrm{rev}(w) \stackrel{\text{def}}{=} \sum_{i=0}^{\alpha-1} 2^i c_{\alpha-1-i}$.

Lemma 1. *Let P_i be a progression with period with $p_i > 2$ and group size 2^{β_i}. Then the emissions of P_i destined to any fixed group occur at time intervals of Δ_i or $\Delta_i + 1$ cycles.*

Proof. The j-th group handles sieve locations

$$\begin{aligned} G_{i,j} &= \left\{ a \ : \ \left\lfloor \mathrm{rev}(a \bmod 2^{\alpha}) \,/\, 2^{\beta_i} \right\rfloor = j \right\} \\ &= \left\{ a \ : \ a \equiv \mathrm{rev}(j) \pmod{2^{\alpha-\beta_i}} \right\} . \end{aligned}$$

By definition, the progression P_i contains the sieve locations

$$P_i = \{ a \ : \ a \equiv r_i \pmod{p_i} \} .$$

Since $2^{\alpha-\beta_i}$ and $p_i > 2$ are coprime, by the Chinese Remainder Theorem we get that

$$P_{i,j} \stackrel{\text{def}}{=} G_{i,j} \cap P_i \ = \ \{ a : a \equiv c_{i,j} \pmod{2^{\alpha-\beta_i} p_i} \} \quad \text{for some } c_{i,j} .$$

Thus, a pair of consecutive $a_1, a_2 \in P_{i,j}$ fulfill $a_2 - a_1 = 2^{\alpha-\beta_i} p_i$. The time difference between the corresponding emissions is $\Delta' \stackrel{\text{def}}{=} \lfloor a_2/s \rfloor - \lfloor a_1/s \rfloor$. If $(a_2 \bmod s) > (a_1 \bmod s)$ then $\Delta' = \lfloor (a_2 - a_1)/s \rfloor = \lfloor 2^{\alpha-\beta_i} p_i / s \rfloor = \Delta_i$, otherwise, $\Delta' = \lceil (a_2 - a_1)/s \rceil = \Delta_i + 1$. □

Note that $\Delta_i \approx \sqrt{p_i}$, by the choice of β_i. The fact that emissions into each group occur in nearly-regular intervals of T_i block cycles (up to a correction of 1 cycle due to modulo s effect) is exploited for efficient implementation of the emitters, as follows.

2.3.2.3 Emitter structure

In the smallish stations, each emitter consists of two counters, as follows.

- Counter A operates modulo $\Delta_i = \lfloor 2^{-\beta_i} p_i \rfloor$ (typically $\langle\!\langle 7 \rangle\!\rangle_{\mathbf{r}}$ $\langle\!\langle 5 \rangle\!\rangle_{\mathbf{a}}$ bits), and keeps track of the time until the next emission of this emitter. It is decremented by 1 (nearly) every clock cycle.
- Counter B operates modulo 2^{β_i} (typically $\langle\!\langle 10 \rangle\!\rangle_{\mathbf{r}}$ $\langle\!\langle 15 \rangle\!\rangle_{\mathbf{a}}$ bits). It keeps track of the β_i most significant bits of the residue class modulo s of the sieve location corresponding to the next emission. It is incremented by $2^{\alpha-\beta_i} p_i \bmod 2^{\beta_i}$ whenever Counter A wraps around. Whenever Counter B wraps around, Counter A is suspended for one clock cycle (this corrects for the modulo s effect).

A delivery pair $(\lfloor \log p_i \rceil, \ell)$ is emitted when Counter A wraps around, where $\lfloor \log p_i \rceil$ is fixed for each emitter. The target bus line ℓ gets β_i of its bits from Counter B. The $\alpha - \beta_i$ least significant bits of ℓ are fixed for this emitter, and they are also fixed throughout the relevant segment of the delivery line so there is no need to transmit them explicitly.

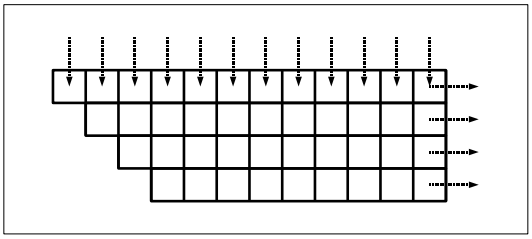

Figure 2.5: Schematic structure of an n-to-m funnel with $n = 12$, $m = 4$.

The physical location of the emitter is near (or underneath) the group of bus lines to which it is attached. The counters and constants need to be set appropriately during device initialization. Note that if the device is custom-built for a specific factorization task then the circuit size can be reduced by hard-wiring many of these values[19]. The combined length of the counters is roughly $\lg p_i$ bits, and with appropriate adjustments they can be implemented using compact ripple adders[20] as in [130].

2.3.2.4 Emitters for tiny progressions

For tiny stations, we use a very similar design. The bus lines are again assigned to residues modulo s in bit-reversed order (indeed, it would be quite expensive to reorder them). This time we choose β_i such that $|G| = 2^{\beta_i}$ is the largest power of 2 that is smaller than p_i. This fixes $\Delta_i = 1$, i.e., an emission occurs every one or two clock cycles. The emitter circuitry is identical to the above, but in this case Counter A has become zero-sized (i.e., a wire), which effectively leaves a single counter of size $\beta_i \approx \lg p_i$ bits.

2.3.3 Implementation of funnels

The smallish stations use *funnels* to compact the sparse outputs of emitters before they are passed to delivery lines (see §2.2.3). We implement these funnels as follows.

An n-to-m funnel ($n \gg m$) consists of a matrix of n columns and m rows, where each cell contains registers for storing a single progression triplet (see Figure 2.5). At every clock cycle inputs are fed directly into the top row, one input per column, scheduled such that the i-th element of the t-th input array is inserted into the i-th column at time $t + i$. At each clock cycle, all values are shifted horizontally one column to the right. Also, each value is shifted one row down (to make room for insertions at the top) if this would not overwrite another value. The t-th output array is read off the rightmost column at time $t + n$.

[19] For sieving the rational side of NFS, it suffices to fix the smoothness bounds. Similarly for the preprocessing stage of Coppersmith's Factorization Factory [47] .

[20] This requires insertion of small delays and tweaking the constant values.

For any $m < n$ there is some probability of "overflow" (i.e., insertion of input value into a full column). Assuming that each input is non-empty with some probability ν independently of the others ($\nu \approx 1/\sqrt{p_i}$; see §2.2.3), the probability that a non-empty input will be lost due to overflow is:

$$\sum_{k=m+1}^{n} \binom{n}{k} \nu^k (1-\nu)^{n-k} (k-m)/k$$

We use funnels with $\langle\!\langle m = 5 \rangle\!\rangle$ rows and $\langle\!\langle n \approx 1/\nu \rangle\!\rangle$ columns. For this choice and within the range of smallish progressions, the above failure probability is less than 0.00011. This certainly suffices for our application (see §1.6.1).

The above funnels have a suboptimal compression ratio $n/m \langle\!\langle \approx 1/5\nu \rangle\!\rangle$, i.e., the probability $\langle\!\langle \approx 1/5 \rangle\!\rangle$ of a funnel output value being non-empty, designated ν', is still rather low. We thus feed these output into a second-level funnel $\langle\!\langle$with $m' = 35$, $n' = 14\rangle\!\rangle_{\mathbf{r}}$, whose overflow probability is less than 0.00016, and whose cost is amortized over many progressions. The output of the second-level funnel is fed into the delivery lines. The combined compression ratio of the two funnel levels is suboptimal only by a factor of $n'/m'\nu' \langle\!\langle \approx 14/34 \cdot 5 = 2 \rangle\!\rangle$, so the number of delivery lines is twice the naive optimum. Note that, these being smallish stations, there is no further complication due to interleaving of adders (see §2.3.1).

2.3.4 Initialization

The device initialization consists of loading the progression states and initial counter values into all stations, and loading instructions into the bus bypass re-routing switches (after mapping out the defects).

The progressions differ between sieving runs, but reloading the device would require significant time (in [185] this became a bottleneck). We can avoid this by noting, as in [74], that the instances of sieving problem that occur in the NFS are strongly related, and all that is needed is to increase each r_i value by some constant value $\tilde{r}_i$ after each run. The $\tilde{r}_i$ values can be stored compactly in DRAM using $\lg p_i$ bits per progression (this is included in our cost estimates) and the addition can be done efficiently using on-wafer special-purpose processors. Since the interval R/s between updates is very large, we don't need to dedicate significant resources to performing the update quickly. For lattice sieving the situation is somewhat different (see §2.3.8).

2.3.5 Cascading the sieves

The following modification is crucial to achieving high parallelism, and affects the high-level architecture and physical partitioning of the device.

Recall that the instances of the sieving problem come in pairs of *rational* and *algebraic* sieves, and we are interested in the a values that passed both sieves (see §1.6). However, the situation is

not symmetric: $U_{\mathbf{a}} \langle\!\langle 2.6 \cdot 10^{10} \rangle\!\rangle_{\mathbf{a}}$ is much larger than $U_{\mathbf{r}} \langle\!\langle = 3.5 \cdot 10^{9} \rangle\!\rangle_{\mathbf{r}}$.[21], leading to much higher progression storage requirements on the algebraic side. Therefore, the cost of the algebraic sieves would dominate the total cost even when s is chosen optimally for each sieve type. Moreover, for 1024-bit composites and the parameters we consider (see §2.4), we cannot make the algebraic-side s as large as we wish because this would exceed the capacity of a single silicon wafer. The following shows a way to address this.

Let s_R and s_A denote the s values of the rational and algebraic sieves respectively. The reason we cannot increase s_A and gain further "free" parallelism is that the bus becomes too wide to fit on a silicon wafer, and the delivery lines become numerous and long (their cost increases as $\tilde{\Theta}(s^2)$).

This cost is incurred because the bus is designed to sieve s_A sieve locations per pipeline stage. But if we first execute the rational sieve, then most sieve locations can be ruled out even before algebraic sieving: all but a small fraction $\langle\!\langle 1.7 \cdot 10^{-4} \rangle\!\rangle$ of the sieve locations do not pass the threshold in the rational sieve,[22] and thus cannot form candidates regardless of their algebraic-side quality.

Accordingly, we make the following change in the design of algebraic sieve. Instead of a wide bus consisting of s_A lines that are permanently assigned to residues modulo s_A, we use a much narrower bus consisting of only $u \langle\!\langle = 32 \rangle\!\rangle_{\mathbf{a}}$ lines, where each line contains a pair (C, L). $L = (a \bmod s_A)$ identifies the sieve location, and C is the sum of $\lfloor \log p_i \rceil$ contributions to a so far. The sieve locations are still scanned in a pipelined manner at a rate of s_A locations per clock cycle, and all delivery pairs are generated as before at the respective units.

The algebraic-side delivery lines are different: instead of being long and "dumb", they are now short and "smart". When a delivery pair $(\lfloor \log p_i \rceil, \ell)$ is generated, ℓ is compared to L for each of the u lines (at the respective pipeline stage) in a single clock cycle. If a match is found, $\lfloor \log p_i \rceil$ is added to the value C of that line. Otherwise (i.e., in the overwhelming majority of cases), the delivery pair is discarded.

At the head of the bus, the algebraic sieve accepts input pairs $(0, a \bmod s_A)$ corresponding to the sieve locations a that passed the rational sieve. To achieve this we wire the outputs of rational sieves to inputs of algebraic sieves, and operate them in a synchronized manner (with the necessary phase shift to account for latency). Due to the mismatch in s values, we connect s_A/s_B rational sieves to each algebraic sieves. Each such cluster of $s_A/s_B + 1$ sieving devices is jointly applied to one single sieve line at a time, in a synchronized manner. To divide the work between the multiple rational sieves, we use interleaving of sieve locations (similarly to the bit-reversal technique of 2.3.2). With buffering to average away congestions, each rational-to-algebraic connection transmits at most one value of size $\lg s_R \langle\!\langle 12 \rangle\!\rangle$ bits per clock cycle — a fairly low bandwidth requirement, which can be handled by external wires connecting separate chips.

[21] $U_{\mathbf{a}}$ and $U_{\mathbf{r}}$ are chosen as to produce a sufficient probability of semi-smoothness for the values over which we are (implicitly) sieving: circa $\langle\!\langle 10^{101} \rangle\!\rangle_{\mathbf{a}}$ vs. circa $\langle\!\langle 10^{64} \rangle\!\rangle_{\mathbf{r}}$.

[22] This is before the additional filtering of cofactor factorization. The fraction is slightly more when considering the rounding inherent in $\lfloor \log p_i \rceil$ and T.

This change greatly reduces the circuit area occupied by the bus wiring and delivery lines; for our choice of parameters, those components becomes insignificant in the algebraic sieve. Also, there is no longer need to duplicate emitters for smallish progressions (except when $p_i < s$). This allows us to use a large $s \langle\!\langle = 32{,}768 \rangle\!\rangle_{\mathbf{a}}$ for the algebraic sieves, thereby reducing their cost to less than that of the rational sieve (see §2.5.1). Hence, we can further increase $U_{\mathbf{a}}$ with little effect on cost, which (due to tradeoffs in the NFS parameter choice) reduces A and B.

2.3.6 Eliminating sieve locations

In the NFS relation collection, we are only interesting in sieve locations a on the b-th sieve line for which $\gcd(a', b) = 1$, where $a' = a - R/2$, since other locations yield duplicate relations. The latter are eliminated by the candidate testing, but the sieving work can be reduced by avoiding sieve locations with $c \,|\, a', b$ for very small c. All major software-based sievers avoid locations with $2 \,|\, a', b$ — this eliminates 25% of the sieve locations. In TWIRL we do the same: first we sieve normally over all the odd lines, $b \equiv 1 \pmod 2$. Then we sieve over the even lines, and consider only odd a' values; since a progression with $p_i > 2$ hits every p_i-th odd sieve location, the only change required is in the initial values loaded into the memories and counters. Sieving of these odd lines takes half the time compared to even lines.

We also consider the case $3 \,|\, a', b$, similarly to the above. Combining the two, we get four types of sieve runs: full-, half-, third- and sixth-length runs, for $b \bmod 6$ in {1,5}, {2,4}, {3} and {0} respectively. Overall, we get a 33% time reduction, essentially for free. It is not worthwhile to consider $c \,|\, a', b$ for $c > 3$ since the additional saving is too small to justify complication of the control logic.

2.3.7 Testing candidates

Having computed approximations of the sum of logarithms $\lambda(a)$ for each sieve location a, we need to identify the resulting candidates, compute the corresponding sets $\{i : a \in P_i\}$, and perform some additional tests (see §1.6). These are implemented as follows.

2.3.7.1 Identifying candidates

In each TWIRL device, at the end of the bus (i.e., downstream for all stations) we place an array of comparators, one per bus line, that identify a values for which $\lambda(a) > T$. In the basic TWIRL design, we operate a pair of sieves (one rational and one algebraic) in unison: at each clock cycle, the sets of bus lines that passed the comparator threshold are communicated between the two devices, and their intersection (i.e., the candidates) are identified. In the cascaded sieves variant, only sieve locations that passed the threshold on the rational TWIRL are further processed by the algebraic TWIRL, and thus the candidates are exactly those sieve locations that passed the threshold in the algebraic TWIRL. The fraction of sieve locations that constitute candidates is very small $\langle\!\langle 2 \cdot 10^{-11} \rangle\!\rangle$.

2.3.7.2 Finding the corresponding progressions

For each candidate we need to compute the set $\{i : a \in P_i\}$, separately for the rational and algebraic sieves. From the context in the NFS algorithm it follows that the elements of this set for which p_i is relatively small can be found easily.[23] It is thus sufficient to find the subset $\{i : a \in P_i\,,\ p_i \text{ is largish}\}$, which is accomplished by having largish stations remember the p_i values of recent progressions and report them upon request.

To implement this, we add two dedicated pipelined channels passing through all the processors in the largish stations. The *lines channel*, of width lg$s$ bits, goes upstream (i.e., opposite to the flow of values in the bus) from the threshold comparators. The *divisors channel*, of width lg$U$ bits, goes downstream. Both have a pipeline register after each processor, and both end up as outputs of the TWIRL device. To each largish processor we attach a *diary*, which is a cyclic list of lg$U$-bit values. Every clock cycle, the processor writes a value to its diary: if the processor inserted an emission triplet $(\lfloor \log p_i \rceil,\, \ell_i,\, \tau_i)$ into the buffer at this clock cycle, it writes the triple $(p_i,\, \ell_i, \tau_i)$ to the diary; otherwise it writes a designated NULL value. When a candidate is identified at some bus line ℓ, the value ℓ is sent upstream through the lines channel. Whenever a processor sees an ℓ value on the lines channel, it inspects its diaries to see whether it made an emission that was added to bus line ℓ exactly z clock cycles ago, where z is the distance (in pipeline stages) from the processor's output into the buffer, through the bus and threshold comparators and back to the processor through the lines channel. This inspection is done by searching the $\langle\!\langle 64 \rangle\!\rangle$ diary entries preceding the one written z clock cycles ago for a non-NULL value $(p_i,\, \ell_i)$ with $\ell_i = \ell$. If such a diary entry is found, the processor transmits p_i downstream via the divisors channel (with retry in case of collision). The probability of intermingling data belonging to different candidates is negligible, and even then we can recover (by appropriate divisibility tests at a later stage).

In the cascaded sieves variant, the algebraic sieve records to diaries only those contributions that were not discarded at the delivery lines. The rational diaries are rather large ($\langle\!\langle 13{,}500 \rangle\!\rangle_{\mathbf{r}}$ entries) since they need to keep their entries a long time — the latency z includes passing through (at worst) all rational bus pipeline stages, all algebraic bus pipeline stages and then going upstream through all rational stations. However, these diaries can be implemented very efficiently as DRAM banks of a degenerate form with a fixed cyclic access order (similarly to the memory banks of the largish stations).

2.3.7.3 Testing candidates

Given the above information, the candidates have to be further processed to account for the various approximations and errors in sieving, and to account for the NFS "large primes" (see §1.6). The first steps (computing the values of the polynomials, dividing out small factors and the diary reports, and testing the size and primality of remaining cofactors) can be effectively

[23] Namely, by finding the small factors (using a suitable algorithm — see §1.2) of $N_{\mathbf{r}}(a', b)$ or $N_{\mathbf{a}}(a', b)$ where b is the line being sieved. This can be done at a later stage and is thus not time-critical.

handled by special-purpose processors and pipelines, which are similar to the division pipeline of [74, Section 4] except that here we have far fewer candidates (see §2.6).

2.3.7.4 Cofactor factorization

The candidates that survived the above steps (and whose cofactors were not prime or sufficiently small) undergo cofactor factorization. This involves factorization of one (and seldom two) integers of size at most $\langle\!\langle 1 \cdot 10^{24} \rangle\!\rangle$. Less than $\langle\!\langle 2 \cdot 10^{-11} \rangle\!\rangle$ of the sieve locations reach this stage (this takes $\lfloor \log p_i \rceil$ rounding errors into consideration), and a modern general-purpose processor can handle each in less than 0.05 seconds. Thus, using dedicated hardware this can be performed at a small fraction of the cost of sieving.

2.3.8 Lattice sieving

The design outlined above is motivated by NFS with line sieving, as described in §1.5.3, which has very large sieve line width $R = 2A$. An important variant is NFS with "special-q" and line sieving (see [129]), which is beneficial in terms of reducing the number of sieve locations tested. However, lattice sieving has very short sieving lines (8192 in [44]), so the natural mapping to the lattice problem as defined here (i.e., lattice sieving by lines) leads to values of A that are too small.

We can adapt TWIRL to efficient lattice sieving as follows. Choose s equal to the width of the lattice sieving region (they are of comparable magnitude); a full lattice line is handled at each clock cycle, and R is the total number of points in the sieved lattice block. The definition (p_i, r_i) is different in this case — they are now related to the vectors used in lattice sieving by vectors (before they are lattice-reduced). The handling of modulo s wrap-around of progressions is now somewhat more complicated, and the emission calculation logic in all station types needs to be adapted. Note that the largish processors are essentially performing lattice sieving by vectors, as they are "throwing" values far into the "future", not to be seen again until their next emission event.

Re-initialization is needed only when the special-q lattices are changed (every $8192 \cdot 5000$ sieve locations in [44]), but it is more expensive. Given the benefits of lattice sieving, it may be advantageous to use faster (but larger) re-initialization circuits and to increase the sieving regions (despite the lower yield); this requires further exploration.

2.3.9 Fault tolerance

Due to its size, each TWIRL device is likely to have multiple local defects caused by imperfections in the VLSI process. To increase the yield of good devices, we make the following adaptations.

Table 2.1: Sieving parameters

Parameter	Meaning	1024-bit	768-bit	512-bit
$R = 2A$	Width of sieve line	$1.1 \cdot 10^{15}$	$3.4 \cdot 10^{13}$	$1.8 \cdot 10^{10}$
B	Number of sieve lines	$2.7 \cdot 10^{8}$	$8.9 \cdot 10^{6}$	$9.0 \cdot 10^{5}$
$U_{\mathbf{r}}$	Rational smoothness bound	$3.5 \cdot 10^{9}$	$1 \cdot 10^{8}$	$1.7 \cdot 10^{7}$
$U_{\mathbf{a}}$	Algebraic smoothness bound	$2.6 \cdot 10^{10}$	$1 \cdot 10^{9}$	$1.7 \cdot 10^{7}$

If any component of a station is defective, we simply avoid using this station. Using a small number of spare stations of each type (with their constants stored in reloadable latches), we can handle the corresponding progressions.

Since our device uses an addition pipeline, it is highly sensitive to faults in the bus lines or associated adders. To handle these, we can add a small number of spare line segments along the bus, and logically re-route portions of bus lines through the spare segments in order to bypass local faults. In this case, the special-purpose processors in largish stations can easily change the bus destination addresses (i.e., ℓ value of emission triplets) to account for re-routing. For smallish and tiny stations it appears harder to account for re-routing, so we just give up adding the corresponding $\lfloor \log p_i \rceil$ values; we may partially compensate by adding a small constant value to the re-routed bus lines. Since the sieving step is intended only as a fairly crude (though highly effective) filter, a few false-positives or false-negatives are acceptable.

2.4 Parametrization

In the following we postulate the parameters and costs of basic components, that were used to obtain the figures in the above sections and the cost estimates in the next section.

2.4.1 NFS parameters

To predict the cost of sieving, we need to estimate the relevant NFS parameters (A, B, $U_{\mathbf{r}}$, $U_{\mathbf{a}}$). The values we used are summarized in Table 2.1. The parameters for 512-bit composites are the same as those postulated for TWINKLE [130] and appear conservative compared to actual experiments [44].[24] For 768-bit and 1024-bit composites, our parameter choice was based on generating concrete NFS polynomials and evaluating their yield. The derivation of these parameters is discussed at depth in Chapter 5.

Since several hardware designs [185, 130, 104, 74] were proposed for the sieving of 512-bit composites, it is instructive to obtain cost estimates for TWIRL with the same problem parameters. We thus assumed the same parameters as in [130, 74].

[24] A strict comparison cannot be done due to the different use of large primes and "special-q".

Where needed for some minor details (e.g., to verify the distribution of tiny primes), we relied on the RSA-155 experiment [44].

2.4.2 Technology parameters

We assume hardware technology parameters which are typical of CMOS VLSI technology circa 2002–3, and are consistent with ITRS 2001 [91]: standard 30cm silicon wafers with 130nm process technology, at an assumed cost of $5,000 per wafer. For 1024-bit and 768-bit composites we will use DRAM-type wafers, which we assume to have a transistor density of 2.8 μm^2 per transistor (averaged over the logic area) and DRAM density of 0.2μm^2 per bit (averaged over the area of DRAM banks). For 512-bit composites we will use logic-type wafers, with transistor density of 2.38μm^2 per transistor and DRAM density of 0.7μm^2 per bit. The clock rate is 1GHz clock rate, which appears realistic with judicious pipelining of the processors.[25]

2.5 Cost estimates

Having outlined the design and specified the problem size and technology parameters, we are ready to estimate the full cost of a hypothetical TWIRL device. We include all of the enhancements described in the preceding sections. While we tried to produce realistic figures, we stress that these estimates are quite rough and rely on many approximations and assumptions. They should only be taken to indicate the order of magnitude of the true cost. We have not done any detailed VLSI design, let alone actual implementation.

2.5.1 Cost of sieving for 1024-bit composites

We assume the aforementioned NFS parameters and set parallelization factors of $s_R = 4{,}096$ on the rational side and $s_A = 32{,}768$ for the algebraic side (using the cascaded sieves variant of §2.3.5).

With this choice, one rational TWIRL device requires 16,000mm^2 of silicon wafer area, or 1/4 of a 30cm silicon wafer. Of this, 76% is occupied by the largish progressions (and specifically, 37% of the device is used for the DRAM banks), 21% is used by the smallish progressions and the rest (3%) is used by the tiny progressions. For the algebraic side we set $s_A = 32{,}768$. One algebraic TWIRL device requires 66,000mm^2 of silicon wafer area — a full wafer. Of this, 94% is occupied by the largish progressions (66% of the device is used for the DRAM banks) and 6% is used by the smallish progressions. Additional parameters of are mentioned throughout §2.2.

The devices are assembled in clusters that consist each of 8 rational TWIRLs and 1 algebraic TWIRL, where each rational TWIRL has a unidirectional link to the algebraic TWIRL over which it transmits 12 bits per clock cycle. A cluster occupies three wafers: one wafer contains an

[25] These technology parameters correspond to the Custom-130-D and Custom-130-L setting of §3.7.1, respectively.

algebraic TWIRL, and each of the other two wafers contains 4 rational TWIRLs. Each cluster handles a full sieve line in R/s_A clock cycles, i.e., 33.4 seconds when clocked at 1GHz. The full sieving involves B sieve lines, which would require 194 years when using a single cluster (after the 33% saving of §2.3.6.) At a cost of $2.9M (assuming $5,000 per wafer), we can build 194 independent TWIRL clusters that, when run in parallel, would complete the sieving task within 1 year.

After accounting for the cost of packaging, power supply and cooling systems, adding the cost of PCs for collecting the data and leaving a generous error margin,[26] it appears realistic that all the sieving required for factoring 1024-bit integers can be completed within 1 year by a device that cost $10M to manufacture.[27] In addition to this per-device cost, there would be an initial Non Recurring Engineering cost on the order of $20M for design, simulation, mask creation, etc.

Further details. We have derived rough estimates for all major components of the design; this required additional analysis, assumptions and simulation of the algorithms. Here are some highlights, for 1024-bit composites with the choice of parameters specified throughout §2.2. A typical largish special-purpose processor is assumed to require the area of $\langle\!\langle 96{,}000 \rangle\!\rangle_{\mathbf{r}}$ logic-density transistors (including the amortized buffer area and the small amount of cache memory, about $\langle\!\langle 14\text{Kbit} \rangle\!\rangle_{\mathbf{r}}$, that is independent of p_i). A typical emitter is assumed to require $\langle\!\langle 2{,}037 \rangle\!\rangle_{\mathbf{r}}$ transistors in a smallish station (including the amortized costs of funnels), and $\langle\!\langle 522 \rangle\!\rangle_{\mathbf{r}}$ in a tiny station. Delivery cells are assumed to require $\langle\!\langle 530 \rangle\!\rangle_{\mathbf{r}}$ transistors with interleaving (i.e., in largish stations) and $\langle\!\langle 1220 \rangle\!\rangle_{\mathbf{r}}$ without interleaving (i.e., in smallish and tiny stations). We assume that the memory system of §2.2.2 requires $\langle\!\langle 2.5 \rangle\!\rangle$ times more area per useful bit than standard DRAM, due to the required slack and area of the cache. We assume that bus wires don't require wafer area apart from their pipelining registers, due to the availability of multiple metal layers. We take the cross-bus density of bus wires to be $\langle\!\langle 0.5 \rangle\!\rangle$ bits per μm, possibly achieved by using multiple metal layers.

Technological challenges. Since the device contains many interconnected units of non-uniform size, designing an efficient layout (which we have not done) is a non-trivial task. However, the number of different unit types is very small compared to designs that are commonly handled by the VLSI industry, and there is considerable room for variations. The mostly systolic design also enables the creation of devices which are larger than the reticle size, using multiple steps of a single (or very few) mask set.

Yield. Using a fault-tolerant design (see §2.3.9), the yield can be made very high and functional testing can be done at a low cost after assembly. Also, the acceptable probability of undetected errors is much higher than that of most VLSI designs.

[26] It is a common rule of thumb to estimate the total cost as twice the silicon cost; to be conservative, we triple it.

[27] With more modern 90nm technology this is reduced to $1.1M; see §2.5.5.

2.5.2 Cost of sieving for 768-bits composites

Using the aforementioned 768-bit NFS parameters, and parallelization factors of $s_R = 1{,}024$ and $s_A = 4{,}096$ (cascaded), we obtain the following.

A rational sieve occupies 1,330mm^2 and an algebraic sieve occupies 4,430mm^2. A cluster consisting of 4 rational sieves and one algebraic sieve can process a sieve line in 8.3 seconds, and 6 independent clusters can fit on a single 30cm silicon wafer.

Thus, a single wafer of TWIRL clusters can complete the sieving task within 95 days. This wafer would cost about $5,000 to manufacture — one tenth of the RSA-768 challenge prize [177].[28]

2.5.3 Cost of sieving for 512-bits composites

The aforementioned 512-bit NFS parameters, assumed for compatibility with [130, 74], do not let us favorably employ the cascaded sieves variation of §2.3.5. We thus set $s_A = s_R = 1{,}024$ without cascading.

A single TWIRL device would have a die size of about 800mm^2, 56% of which are occupied by largish progressions and most of the rest occupied by smallish progressions. It would process a sieve line in 0.018 seconds, and can complete the sieving task within 6 hours.

In comparison, for the same NFS parameters TWINKLE would require 1.8 seconds per sieve line, the FPGA-based design of [104] would require about 10 seconds and the mesh-based design of [74] would require 0.36 seconds. To provide a fair comparison to TWINKLE and [74], we should consider a single wafer full of TWIRL devices running in parallel. Since we can fit 79 of them, the effective time per sieve line is 0.00022 seconds.

Thus, in factoring 512-bit composites the basic TWIRL design is about 1,600 times more cost effective than the best previously published design [74], and 8,100 times more cost effective than TWINKLE. Adjusting the NFS parameters to take advantage of the cascaded-sieves variant (see §2.3.5) would further increase this gap. However, even when using the basic variant, a single wafer of TWIRLs can complete the sieving for 512-bit composites in under 10 minutes.

2.5.4 Asymptotic behavior for larger composites

For largish progressions, the amortized cost per progression is $\Theta(s^2(\log s)/p_i + \log s + \log p_i)$ with small constants (see §2.2.2). For smallish progressions, the amortized cost is $\Theta(\,(s/\sqrt{p_i} + 1)\,(\log s + \log p_i)\,)$ with much larger constants (see §2.2.3). For a serial implementation (PC-based or TWINKLE), the cost per progression is clearly $\Omega(\log p_i)$. This means that asymptotically we can choose $s = \tilde{\Theta}(\sqrt{U})$ to get a speed advantage of $\tilde{\Theta}(\sqrt{U})$ over serial implementations, while maintaining the small constants. Indeed, we can keep increasing s essentially for free until the

[28] Needless to say, this disregards an initial cost of about $20M. This initial cost can be significantly reduced by using older technology, such as 0.25μm process, in exchange for some decrease in sieving throughput.

area of the largish processors, buffers and delivery lines becomes comparable to the area occupied by the DRAM that holds the progression triplets.

For sufficiently large composites, it becomes beneficial to reduce the amount of DRAM used for largish progressions by storing only the prime p_i, and computing the rest of the progression triplet values on-the-fly in the special-purpose processors (this requires computing the roots modulo p_i of the relevant NFS polynomial).

If the device would exceed the capacity of a single silicon wafer, then as long as the bus itself is narrower than a wafer, we can (with appropriate partitioning) keep each station fully contained in some wafer; the wafers are connected in a serial chain, with the bus passing through all of them. Such high-bandwidth interconnects are non-trivial but feasible; the unidirectional, latency-insensitive data flow along the bus (except for the diaries) helps in this respect.

2.5.5 Scaling with technology

In terms of data flow, TWIRL uses an enormous bandwidth both along the main pipeline (between stations) and across it (along delivery lines). It is thus inherently a single-wafer design: if we attempt to partition it into several chips (whether ASIC or FPGA), its performance will be greatly reduced due to the limited throughput of the connections between the chips. Consequentially, for 1024-bit it is presently necessary to use a sub-optimal choice of the algebraic factor base bound B_{a} in order to fit all progressions into a single wafer, even when using DRAM storage. A consequence of this is that presently, the cost of TWIRL for 1024-bit (or larger) composites decreases faster than naively implied by technological improvement in transistor speed and size (i.e., Moore's law).

For example, moving from 130nm VLSI process to 90nm VLSI process the circuit area shrinks by a factor of 4, but the cost of the device (after re-optimizing the various parameters) decreases by a factor of 9, to just 1.1 US$\$\times$years. See §5.5.3 for further discussion.

2.6 Comparison to previous works

TWINKLE. The new device shares with TWINKLE (see §1.6.4) the property of time-space reversal compared to traditional sieving. TWIRL is evidently much faster than TWINKLE, as the two have comparable clock rates but the latter checks one sieve location per clock cycle whereas the former checks thousands. None the less, TWIRL is smaller than TWINKLE — this is due to the efficient parallelization and the use of compact DRAM storage for the largish progressions (it so happens that DRAM cannot be efficiently implemented on GaAs wafers, which are used by TWINKLE). We may consider using TWINKLE-like optical analog adders instead of electronic adder pipelines, but constructing a separate optical adder for each residue class modulo s would entail practical difficulties, and does not appear worthwhile as there are far fewer values to sum.

Relation collection without sieving. Bernstein [22] proposes to completely replace sieving by memory-efficient smoothness testing methods, such as the Elliptic Curve Method factoring algorithm, in order to reduces the asymptotic throughput cost of the linear algebra step. However, these asymptotic figures hide significant factors; based on current experience, for 1024-bit composites it appears unlikely that memory-efficient smoothness testing would rival the practical performance of traditional sieving, let alone that of TWIRL.

Mesh-based sieving. Compared to previous sieving devices, both mesh-based sieving (see §1.6.6) and TWIRL achieve a speedup factor of $\tilde{\Theta}(\sqrt{U})$. However, there are significant differences in scalability and cost: TWIRL is 1,600 times more efficient for 512-bit composites, and ever more so for bigger composites or when using the cascaded sieves variant (see §2.5.3, 2.3.5). Some reasons for the discrepancy are as follows.[29]

The mesh-based sorting of [74] is effective in terms of *latency*, which is why it was appropriate for the Bernstein's matrix-step device [22] where the input to each invocation depended on the output of the previous one. However, for sieving we care only about *throughput*. Disregarding latency leads to smaller circuits and higher clock rates. For example, TWIRL's delivery lines perform trivial one-dimensional unidirectional routing of values of size $\langle\!\langle 12+10 \rangle\!\rangle_{\mathbf{r}}$ bits, as opposed to complicated two-dimensional mesh sorting of progression states of size $\langle\!\langle 2 \cdot 31.7 \rangle\!\rangle_{\mathbf{r}}$. For the algebraic sieves the situation is even more extreme (see §2.3.5).

In the design of [74], the state of each progression is duplicated $\lceil \tilde{\Theta}(U/p_i) \rceil$ times (compared to $\lceil \tilde{\Theta}(\sqrt{U/p_i}) \rceil$ in TWIRL) or handled by other means; this greatly increases the cost. For the primary set of design parameters suggested in [74] for factoring 512-bit numbers, 75% of the mesh is occupied by duplicated values even though all primes smaller than 2^{17} are handled by other means, namely a separate division pipeline that tests potential candidates identified by the mesh (using over 12,000 expensive integer division units). Moreover, this assumes that the sums of $\lfloor \log p_i \rfloor$ contributions from the progressions with $p_i > 2^{17}$ are sufficiently correlated with smoothness under all progressions; it is unclear whether this assumption scales.

TWIRL's handling of largish primes using DRAM storage greatly reduces the size of the circuit when implemented using current VLSI technology (90 DRAM bits vs. about 2500 transistors in [74]).

If the device must span multiple wafers, the inter-wafer bandwidth requirements of our design are high, but still much lower than that of [74] (as long as the bus is narrower than a wafer), and there is no algorithmic difficulty in handling the long latency of cross-wafer lines. Moreover, connecting wafers in a chain may be easier than connecting them in a 2D mesh, especially in regard to cooling and faults.

Other devices. All of the remaining sieving devices surveyed in §1.6 have a significantly higher cost for 512-bit composites (where known), and their scalability (where possibly to evaluate) is worse than the above.

[29] These were mostly addressed in the improved design [76] subsequent to the publication of TWIRL [187]; see §6.3.

Chapter 3

A mesh-based architecture for the NFS linear algebra step

3.1 Overview

Turning our attention from the NFS sieving step to the other major step in the NFS algorithm, this chapter addresses an approach to special-purpose devices for the NFS linear algebra step. Here we are concerned with architectures based on a two-dimensional mesh or torus, realized as electronic VLSI circuits.

We begin by evaluating such a proposal by Bernstein [22], for a simple architecture based on mesh sorting. We show that, despite the attractive asymptotic behavior, the concrete cost of this proposal is prohibitive.

We then present an alternative architecture based on mesh routing. The resulting design has improved efficiency, fault tolerance and adaptivity to technological consideration. Its optimized cost, about 67,000 times lower than that of [22]'s circuit, brings the linear algebra step to well within practical reach for 1024-bit composites.

An alternative approach, with notable addition advantages, is portrayed in Chapter 4.

3.2 Estimating the cost of Bernstein's circuits

We begin by concretely evaluating the cost of Bernstein's proposed circuit [22], concluding that its cost is prohibitive. This will motivate our alternative design in subsequent sections; moreover, it will provide context for the identification and removal of various cost bottlenecks.

We consider the case of 1024-bit composites (see §1.3.2). To provide a best-case estimate we choose a set of NFS parameters chosen for minimized throughput cost (in accordance with [22]'s

stated goal): the *throughput-optimized matrix* parameter set from §5.4.1.2. We then derive a rough prediction of the associated costs when the mesh is implemented by custom hardware using modern VLSI technology. In this section we use the circuit exactly as described in [22]; we assume familiarity with that approach and refer the reader to [22] for its details. The next subsections will describe an improved architecture, incorporating (among others) the improvements listed as future plans in [22].

The circuits of [22] employ Wiedemann's original algorithm [216], which is a special case of block Wiedemann with blocking factor K=1 (see §1.7.1). Given an input consisting of a matrix $A \in \mathrm{GF}(2)^{D \times D}$ with average column weight h and a vector $\vec{v} \in \mathrm{GF}(2)^D$, the algorithm requires a mesh of $m \times m$ nodes where $m^2 > hD + 2D$. By assumption, $h \approx 100$ and $D \approx 4 \cdot 10^7$ so we may choose $m = 63256$. To execute the sorting-based algorithm, each node consists mainly of 3 registers of $\lceil \log_2(4 \cdot 10^7) \rceil = 26$ bits each, a 26-bit compare-exchange element (in at least half of the nodes), and some logic for tracking the current stage of the algorithm. Input, namely the nonzero elements of A and the initial vector $\vec{v}$, is loaded just once so this can be done serially. The mesh computes the vectors $A^k\vec{v}$ by repeated matrix-by-vector multiplication, and following each such multiplication it calculates the inner product $\vec{u}(A^k\vec{v})$ and outputs this single bit.

In terms of transistor count, assuming 8 transistors per stored bit[1], storage amounts to 624 transistors per node. Accounting for the logic for additional overheads such as a clock distribution network, we shall assume an average of 2000 transistors per node for a total of roughly $8.0 \cdot 10^{12}$ transistors in the mesh.

Using 130nm CMOS VLSI process on 30cm silicon wafers circa 2002 (i.e., the Custom-130-L setting of §3.7.1), the complete mesh would occupy the area of 273 full silicon wafers.[2] Optimistically assuming $5,000 per wafer, the construction cost would be US$1.4M, plus several million US$ for the initial Non Recurring Engineering costs.

Inter-chip communication. The matter of inter-chip communication is problematic. The mesh as a whole needs very few external lines (serial input, 1-bit output, clock, and power). However, a chip consisting of $s \times s$ nodes has $4s - 4$ nodes on its edges, and each of these needs two 26-bit bidirectional links with its neighbor on an adjacent chip, for a total of about $2 \cdot 2 \cdot 26 \cdot 4s = 416s$ connections. Moreover, such connections typically do not support the full 1GHz clock rate, so to achieve the necessary bandwidth we will need about 4 times as many connections: $1664s$. Standard wiring technology cannot provide such enormous density, though emerging technologies do aim to achieve it.

For concreteness, we shall complete the evaluation using one such emerging approach. 'Flip-chip" technologies allow direct connections between chips that are placed face-to-face, at a density of 277 connections per mm^2 (i.e., 60μs array pitch). We cut each wafer into the shape of a cross, and arrange the wafers in a two-dimensional grid with the arms of adjacent wafers in full overlap and facing each other. The central square of each cross-shaped wafer contains mesh nodes, and the arms are dedicated to inter-wafer connections. Simple calculation shows that with the above

[1] A single-bit register (D-type edge-triggered flip-flop) in standard CMOS logic.
[2] See [131] for some further details.

connection density, if 40% of the (uncut) wafer area is used for mesh nodes then there is sufficient room left for the connection pads and associated circuitry. This disregards the issues of delays (mesh edges that cross wafer boundaries are realized by longer wires and are thus slower than the rest), and of the defects which are bound to occur. To address these, adaptation of the algorithm is needed.

Assuming that these technological and algorithmic issues are surmountable, the inter-wafer communication entails a cost increase by a factor of about 3, to US$4.1M.

Throughput cost. According to [22, Section 4], a matrix-by-vector multiplication consists of, essentially, three sort operations on the $m \times m$ mesh. Each sort operation takes $8m$ steps, where each step consists of a compare-exchange operation between 26-bit registers of adjacent nodes. Thus, multiplication requires $3 \cdot 8m \approx 1.52 \cdot 10^6$ steps. Assuming that each step takes a single clock cycle at a 1GHz clock rate, we get a throughput of 659 multiplications per second.

Basically, Wiedemann's algorithm requires $2D$ multiplications. Alas, the use of blocking factor $K = 1$ entails additional costs that increase the number of necessary multiplications to roughly $20D$ (see §1.7.2). Thus, the expected total running time is roughly $20 \cdot 4 \cdot 10^7/659 \approx 1\,210\,000$ seconds, or 14 days. The throughput cost is thus $5.10 \cdot 10^{12}$ US$ × seconds.

If we increase the blocking factor from 1 to over 32 and handle the multiplication chains sequentially on a single mesh, then only $2D$ multiplications are needed.[3] In this case the time decreases to 34 hours, and the throughput cost decreases to $5 \cdot 10^{11}$ US$ × seconds.

Since the device consists solely of logic circuitry that is active all the time, it would have very high power consumption (equiv., heat dissipation) which may limit the node density and clock rate of the device.[4]

3.3 Basic routing-based architecture

We proceed to present an alternative mesh-based architecture. This design performs a single routing operation per multiplication, compared to three sorting operations (where even a single sorting operation is slower than routing). It features a reduced cost, improved fault tolerance and very simple local control. Moreover, its inherent flexibility allows further improvements, as discussed in subsequent sections, yielding a total reduction in throughput cost by a factor of about 67,000 compared to Bernstein's circuit (as evaluated above, for 1024-bit composites). In this section, we describe the basic approach.

For simplicity assume that each of the D columns of the matrix has weight exactly h (here $h = 100$), and that the nonzero elements of A are uniformly distributed (both assumptions can be easily relaxed, as done in §4.2.4). Let $m = \sqrt{D \cdot h}$. We divide the $m \times m$ mesh into D blocks of

[3] [22] considers this but claims that it will not change the cost of computation; that is true only up to constant factors.

[4] Similar considerations arose, and were addressed, for TWINKLE [130].

size $\sqrt{h} \times \sqrt{h}$. Let S_i denote the i-th block in row-major order ($i \in \{1, \ldots, D\}$), and let t_i denote the node in the upper left corner of S_i. We say that t_i is the *target of the value* i. Each node holds two $\log_2 D$-bit values, $Q[i]$ and $R[i]$. Each target node t_i also contains a single-bit value $P[i]$. For repeated multiplication of A and $\vec{v}$, the mesh is initialized as follows: the i-th entry of $\vec{v}$ is loaded into $P[i]$, and the row indices of the nonzero elements in column $i \in \{1, \ldots, D\}$ of A are stored (in arbitrary order) in the $Q[\cdot]$ of the nodes in S_i. Each multiplication is performed thus:

1. For all i, broadcast the value of $P[i]$ from t_i to the rest of the nodes in S_i (this can be accomplished in $2\sqrt{h} - 2$ steps).
2. For all i and every node j in S_i: if $P[i] = 1$ then $R[j] \leftarrow Q[j]$, else $R[j] \leftarrow$ NIL (where NIL is some distinguished value outside $\{1, \ldots, D\}$).
3. $P[i] \leftarrow 0$ for all i
4. Invoke a mesh-based packet routing algorithm on the $R[\cdot]$, such that each non-NIL value $R[j]$ is routed to its target node $t_{R[j]}$. Each time a value i arrives at its target t_i, discard it and flip $P[i]$.

After these steps, $P[\cdot]$ contain the result of the multiplication, and the mesh is ready for the next multiplication. As before, in the inner product computation stage of the Wiedemann algorithm, we need only compute $\vec{u}A^k\vec{v}$ for some vector $\vec{u}$, so we load the i-th coordinate of $\vec{u}$ into node t_i during initialization, and compute the single-bit result $\vec{u}A^k\vec{v}$ inside the mesh during the next multiplication.

A physical realization of the mesh will contain many local faults (especially for devices that are wafer-scale or larger, as discussed below). In the routing-based mesh, we can handle local defects by algorithmic means as follows. Each node shall contain 4 additional state bits, indicating whether each of its 4 neighbors is "disabled". These bits are loaded during device initialization, after mapping out the defects. The compare-exchange logic is augmented such that if node i has a "disabled" neighbor in direction Δ then i never performs an exchange in that direction, but always performs an unconditional exchange in the two directions orthogonal to Δ. This allows us to "close off" arbitrary rectangular regions of the mesh, such that values that reach a "closed-off" region from outside are routed along its perimeter (clockwise or anti-clockwise, depending on the parity of their arrival time and place). We add a few spare nodes to the mesh, and manipulate the mesh inputs such that the spare effectively replace the nodes of the in closed-off regions. We conjecture that the local disturbance caused by a few small closed-off regions will not have a significant effect on the routing performance.

Going back to the cost evaluation, we see that replacing the sorting-based mesh with a routing-based mesh reduces time by a factor of $3 \cdot 8/2 = 12$. Also, note that the $Q[\cdot]$ values are used just once per multiplication, and can thus be stored in slower DRAM-type cells in the vicinity of the node. DRAM cells are much smaller than edge-triggered flip-flops, since they require only

one transistor and one capacitor per bit. Moreover, the regular structure of DRAM banks allows for very dense packing.[5] For simplicity, we ignore the circuitry needed to retrieve the values from DRAM — this can be done cheaply by temporarily wiring chains of adjacent $R[\cdot]$ into shift registers. In terms of circuit size, we effectively eliminate two of the three large registers per node, and some associated logic, so the routing-based mesh is about 3 times cheaper to manufacture.

Thus far, we gain a reduction of a factor $3 \cdot 12 = 36$ in the throughput cost compared to the sorting-based approach. Additional gains will be described in §3.5 and §3.6.

3.4 Choice of routing algorithm

3.4.1 Criteria and alternatives

The above leaves open the choice of a routing algorithm. Many applicable candidates exist[6]. However, to minimize hardware cost, we focus our attention on algorithms for a mesh topology, under the restrictive *one-packet communication model*[7], as defined by Schnorr and Shamir [183], in which at each step every node holds at most one packet, and the only operation allowed is a local compare-and-exchange of packets between neighboring nodes. Moreover, we seek a simple, periodic schedule (cf. Kutyłowski et al. [112]). These constraints are are crucial for compactness and feasibility of the circuit: maximal locality of wiring (and hence scalability), absence of storage buffers, and simple control logic.

The above model rules out most known algorithms. For example, the time-optimal bounded-queue MIMD algorithm of Leighton et al. [124], as well as its refinements (see [84, §3.1]), perform simultaneous communication along all edges and employ buffers. Similarly, in the *hot-potato routing model*, time-optimal algorithms are known for the torus (Feige et al. [61]) and mesh (Kaklamanis et al. [96]), but such algorithms communicate on all edges at each step, and require deep high-latency logic (even when the *one-pass* property of [61] holds). Note that in our architecture the same routing problem arises repeatedly, and thus the off-line routing model is in principle applicable; however, we are not aware of a satisfactory solution for a succinct local representation of the precomputed routing schedule.

Since our routing task represents addition in GF(2), the routing problem has the following unusual property: pairwise packet annihilation is allowed. That is, pairs of packets with identical destinations may be "canceled out" without affecting the result of the computation. This relaxation can greatly reduce the congestion caused by multiple packets converging to the same destination, and invalidates many lower-bound proofs.

We thus suggest a new routing algorithm, which fits in the desired model and whose empirical properties are essentially optimal.

[5] For example, using the Custom-130-L setting of §3.7.1, in a large banks of embedded DRAM (which are shared by many nodes in their vicinity), the amortized chip area per DRAM bit is about $0.7\mu\text{m}^2$, compared to $2.38\mu\text{m}^2$ per transistor or $19\mu\text{m}^2$ per flip-flop.

[6] See Grammatikakis et al. [84] for a (slightly outdated) survey.

[7] The terminology follows Sibeyn [194].

3.4.2 Clockwise transposition routing

The algorithm, which we call *clockwise transposition routing*, has an exceptionally simple control structure which consists of repeating 4 steps. Each step involves compare-exchange operations on pairs of neighboring nodes, such that the exchange is performed iff it reduces the distance-to-target of the non-NIL value (out of at most 2) that is farthest from its target along the relevant direction.

Formally, the compare-exchange is defined as follows. Consider adjacent cells at columns j, $j+1$ which hold packets destined to columns j_0, j_1 respectively (either of which may be NIL if the corresponding cell is empty). The exchange is performed if one of the following holds:

- Single packet: $j_1 = \text{NIL}$ and $j_0 > j$, or $j_0 = \text{NIL}$ and $j_1 < j+1$.
- Farthest first along compared direction: $j_0, j_1 \neq \text{NIL}$ and $j_0 \geq j_1$.

The analogous rule holds for packets destined to vertically adjacent cells. When a packet reaches its destination, it is consumed (removed from the mesh) by flipping the value of the target's $P[i]$ register.

In addition, we use the following rule, justified by the modulo 2 arithmetics: whenever two identical packets are compared, both are annihilated.[8] The schedule of the compare-exchange operations is as follows:

- In step $t \equiv 0 \pmod 4$, for every column j and odd row i, compare the cell at (i,j) to the cell at $(i-1,j)$, i.e., above it (if any).
- In step $t \equiv 1 \pmod 4$, for every row i and odd column j, compare the cell at (i,j) to the cell at $(i,j+1)$, i.e., to its right (if any).
- In step $t \equiv 2 \pmod 4$, for every column j and odd row i, compare the cell at (i,j) to the cell at $(i+1,j)$, i.e., below it (if any).
- In step $t \equiv 3 \pmod 4$, for every row i and odd column j, compare the cell at (i,j) to the cell at $(i,j-1)$, i.e., to its left (if any).

Note that each cell performs compare-exchange with its 4 neighbors in cyclic clockwise or anti-clockwise order, where the direction and phase depend on its location; hence the name *clockwise transposition routing*. This schedule can be implemented in a pipelined fashion by interleaving 4 routing operations, thereby eliminating the cost of state-keeping and control.

Due to the simplicity and locality of all operations and their schedule, clockwise transposition routing is a very attractive choice for the NFS linear algebra step implementation in special-purpose hardware.

[8] With a blocking factor $K > 1$, as used below, each packet contains a payload of a vector over GF(2) and the packets are merged by adding (XORing) their payloads rather than being deleted.

While no theoretical analysis of this algorithm's performance is known, experimentally its average-case performance appears close to optimal: for randomly filled $m \times m$ meshes, and m sufficiently large, the running time of the algorithm seems to be close to the trivial lower bound $2m - 2$ with overwhelming probability. We have simulated the algorithm on numerous inputs of sizes up to $13\,000 \times 13\,000$, drawn from a distribution mimicking that of the meshes arising in NFS (as well as the simple distribution that puts a random value in every node). In all such runs (except for very small meshes), we have not observed even a single case where the running time exceeded $2m$ steps, which is just 2 steps from optimal; some typical simulations are demonstrated at [202]. However, it is known to livelock on certain pathological inputs as explained below.

This algorithm is a generalization of odd-even transposition sort, with a schedule that is identical to the "2D-bubblesort" algorithm of Ierardi [89] but with different compare-exchange elements. The change from sorting to routing is indeed quite beneficial, as [89] shows that 2D-bubblesort is considerably slower than the observed performance of our clockwise transposition routing. The new algorithm appears to be much faster than the $8m$ sorting algorithm (due to Schimmler [180]) used by Bernstein [22], and its local control is very simple compared to the complicated recursive algorithms of Schnorr and Shamir [183] that achieve the $3m$-step lower bound on mesh sorting. Despite the apparent simplicity, the algorithm is very sensitive to the rule and schedule of the compare-exchange elements; many apparently innocuous changes cause it to misbehave dramatically (in part, analogously to the cases studied in [89]). For example, changing the compare-exchange rule from "farthest-first" to "nearest-first" leads to frequent deadlocks.

3.4.3 Pathologies

While the above algorithm is highly attractive in terms of performance, it eventually turned out to be incomplete — in the sense of having livelock configurations for which it never terminates (i.e., it does not deliver all packets in finite time). The simplest example is "rotation" of the 4×4 mesh, defined as follows (the matrix entries give the row and column indices of the packet's target):

(3,0)	(2,0)	(1,0)	(0,0)
(3,1)	(2,1)	(1,1)	(0,1)
(3,2)	(2,2)	(1,2)	(0,2)
(3,3)	(2,3)	(1,3)	(0,3)

On this input, the compare-exchange rule maintains the following invariant: in each row all packets contain the same destination column, and vice versa. Hence the exchange will always be performed. We can thus easily track each packet and see that it never reaches its destination. Indeed, the original state is restored after 16 steps.

More generally, we can define an $m \times m$ "rotation" as the routing problem in which the packet initially at (i, j) is destined to $(m - 1 - j, i)$. Then for m which is a multiple of 4, clockwise transposition routing will enter a loop of length $4m$.

One may wonder why these livelocks failed to occur in our numerous computer experiments. Intuitively, this can be explained as follows. All known livelock configurations are unstable, in

the sense that when they are embedded in a larger mesh, interaction with a packet "passing by" is likely to break their symmetries. Moreover, the observed livelock configurations cannot "travel" in a larger mesh — the constituent packets and their destinations must be very close to begin with. Thus, we may expect that in a large mesh there is a low probability that a local livelock configuration will occur and survive.

In addition to livelocks, there are pathological inputs that do terminate but take significantly more than $2m$ steps; one example is the "transpose" routing problem, where for every i, j the packet initially at (i, j) is destined to (j, i) (see [202]).[9]

A full theoretical characterization of these pathologies and their probability of occurrence, as well as finding an equally efficient algorithm that avoids them, remains an open problem.

Addressing pathologies. In light of the above, this algorithm is of limited use as a general-purpose routing algorithm. However, in our setting it performs exceptionally well and the pathologies are so rare as to never occur in extensive simulation. Since (in the case of $K > 1$) we perform the same few routing operations repeatedly with different payloads, one can simply verify beforehand that these specific inputs are not pathological (and perturb the matrix by permutation if they are). Moreover, in [70][73] we describe an efficient way to detect pathologies (should they occur), and resolve them using a negligible amount of additional circuitry. Lastly, if pathologies occur sufficiently seldom, we can simply assume that the routing has finished within some time bound close to the optimum, and apply the fault detection method of §4.3 to catch and recover from violations of this assumption.

3.5 Improvements

We now tweak the routing-based circuit design to gain additional cost reductions. Compared to the sorting-based design (see §3.2), these will yield a (constant-factor) improvement by several orders of magnitudes. Moreover, it shows that even for 1024-bit composites, the cost of massive computational parallelization can be made negligible compared to the cost of the RAM needed to store the input, and thus the speed advantage is gained essentially for free.

Increased target density. The first improvement follows from increasing the density of targets. Let ρ denote the average number of $P[\cdot]$ registers per node. In the above scheme, $\rho = h^{-1} \approx 1/100$. The total number of $P[\cdot]$ registers is fixed at D, so if we increase ρ the number of mesh nodes decreases by $h\rho$. However, we no longer have enough mesh nodes to route all the hD nonzero entries of A simultaneously. We address this by partially serializing the routing process, as follows. Instead of storing one matrix entry $Q[\cdot]$ per node, we store $h\rho$ such values per node: for $\rho \geq 1$, each node j is "in charge" of a set of ρ matrix columns $C_j = \{c_{j,1}, \ldots, c_{j,\rho}\}$, in the sense that node j contains the registers $P[c_{j,1}], \ldots, P[c_{j,\rho}]$, and the nonzero elements of A in columns $c_{j,1}, \ldots, c_{j,\rho}$. To carry out a matrix-by-vector multiplication we perform $h\rho$ iterations, where each iteration

[9] This test case was suggested to us by Uriel Feige.

consists of retrieving the next such nonzero element (or skipping it, depending on the result of the previous multiplication) and then performing clockwise transposition routing as before.

Blocking. The second improvement follows from using block Wiedemann with a blocking factor $K \gg 1$ (see §1.7.1). Besides reducing the number of multiplications by a factor of roughly 10 (see §1.7.1, §3.2), this produces an opportunity for amortizing the cost of routing, as follows. Recall that in block Wiedemann, we need to (twice) perform K multiplication chains of the form $A^k\vec{v}_i$, for $i = 1, \ldots, K$ and $k = 1, \ldots, D/K$. The idea is to perform several chains in parallel on a single mesh, reusing most resources (in particular, the storage taken by A). For simplicity, we will consider handling all K chains on one mesh. In the routing-based circuits described so far, each node emitted at most one message per routing operation — a matrix row index, which implies the address of the target cell. The information content of this message (or its absence) is a single bit. Instead, we shall attach K bits of information to this message: $\lg D$ bits for the row index, and K bits of "payload", one bit per multiplication chain.

The full algorithm. Combining the two generalizations gives the following algorithm, for $0 < \rho \leq 1$ and integer $K \geq 1$. The case $0 < \rho < 1$ requires distributing the entries of each matrix column among several mesh nodes, as in §3.3, but its cost is similar.

Let $\{C_j\}_{j \in \{1,\ldots,D/\rho\}}$ be a partition of $\{1, \ldots, D\}$, $C_j = \{c : (j-1)\rho \leq c - 1 < j\rho\}$. Each node $j \in \{1, \ldots, D/\rho\}$ contains single-bit registers $P_i[c]$ and $P'_i[c]$ for all $i = 1, \ldots, K$ and $c \in C_j$, and a register R_j of size $\lg D + K$. Node j also contains a list $Q_j = \{(r, c) \mid A_{r,c} = 1, c \in C_j\}$ of the nonzero matrix entries in the columns C_j of A, and an index I_j into C_j. Initially, load the vectors $\vec{v}_i$ into the $P_i[\cdot]$ registers. Each multiplication is then performed thus:

1. For all i and c, $P'_i[c] \leftarrow 0$. For all j, $I_j \leftarrow 1$.
2. Repeat $h\rho$ times:
   (a) For all j: $(r, c) \leftarrow Q_j[I_j]$, $I_j \leftarrow I_j + 1$, $R[j] \leftarrow \langle r, P_1[c], \ldots, P_K[c]\rangle$.
   (b) Invoke the clockwise transposition routing algorithm on the $R[\cdot]$, such that each value $R[j] = \langle r, \ldots\rangle$ is routed to the node t_j for which $r \in C_j$.
   During routing, whenever a node j receives a message $\langle r, p_1, \ldots, p_K\rangle$ such that $r \in C_j$, it sets $P'_i[r] \leftarrow P'_i[r] \oplus p_i$ for $i = 1, \ldots, K$ and discards the message. Moreover, whenever identically-targeted packets $\langle r, p_1, \ldots, p_K\rangle$ and $\langle r, p'_1, \ldots, p'_K\rangle$ in adjacent nodes are compared, they are combined: one is annihilated and the other is replaced by $\langle r, p_1 \oplus p'_1, \ldots, p_K \oplus p'_K\rangle$.
3. $P_i[c] \leftarrow P'_i[c]$ for all i and c.

After these steps, $P_i[\cdot]$ contain the bits of $A^k\vec{v}_i$ and the mesh is ready for the next multiplication. We need to compute and output the inner products $\vec{u}_j(A^k\vec{v}_i)$ for some vectors $\vec{u}_1, \ldots, \vec{u}_K$, and this computation should be completed before the next multiplication is done. In general, this seems to require $\Theta(K^2)$ additional wires between neighboring mesh nodes and additional registers.

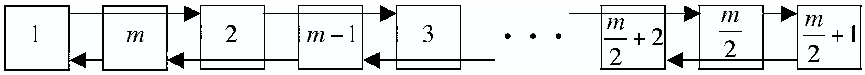

Figure 3.1: Realizing a torus in a flat array.

However, usually the $\vec{u}_j$ are chosen to have weight 1 or 2, so the cost of computing these inner products can be kept very low. Also, note that the number of routed messages is now doubled, because previously only half the nodes sent non-NIL messages. However, empirically it appears that the clockwise transposition routing algorithm handles the full load without any slowdown.

3.6 Further improvement

The following briefly outlines several additional enhancements we have devised for the routing-based architecture; see our publication [70] for details. We also mention a variant by Geiselmann and Steinwandt which improves scalability.

Torus topology. Changing the topology from a mesh to a torus halves the diameter of the graph and reduces the bottleneck effect at the center of the mesh, so we can expect improved performance. The clockwise transposition routing algorithm can be adapted to the torus[10], and its performance remains close to optimal: the empirically observed running time is halved.[11] Since the underlying physical circuit remains mesh-like, we need to emulate a torus by interleaving (see Figure 3.1), which requires longer wires; strictly speaking, the one-packet model is thus violated. However, in practice and for the parameters considered, the additional cost for the longer wires (i.e., additional metal layers in the VLSI process) is much smaller than a factor of 2, so this improvements results in a net benefit.

Parallel tori. The above can be further improved by an additional factor of ≈ 2, by using four interleaved but independent sub-tori of size $m/2 \times m/2$, instead of one $m \times m$ torus.[12] One sub-torus consists of the cells at even rows and even columns; another consists of the cells at even rows and odd columns, and so on. In each sub-torus, the expected source-to-destination distance is merely $m/4$ as opposed to $m/2$ in the single torus (and the maximum distance is similarly halved). Taking into account the interleaving already done in order to realize the torus topology, each edge now physically spans a distance of 3 cells. The cost/benefit situation is comparable to that of the move from the mesh to the torus (but decreases exponentially if we use higher interleaving, so this seems to the maximum beneficial extension). Further local circuitry is now needed in order to initially inject packets into the correct torus.

Refilling. For the relevant parameter choices (and specifically for large values of ρ), each cell has several packets that have to be routed. Thus far, this was handled by iterating the routing

[10] This was first proposed by Geiselmann and Steinwandt [76] in the context of mesh-based *sieving*.
[11] This holds even though the *expected* travel distance, for random inputs, has decreased by a factor of just 4/3.
[12] This too was first proposed in [76] in the context of mesh-based *sieving*.

algorithm several times: at each iteration a set of packets is injected into the mesh and all of them are routed to their destination; only then the next set is injected. To improve performance, we can inject packets before the mesh has been fully cleared. The choice of injection criteria is crucial to realizing the benefit; some natural rules results in congestion and, subsequently, a routing throughput that is lower than the original scheme.

The combined total improvement from the parallel tori topology and refilling is a factor of approximately 4 over the results of §3.5.

Distributed version. In [76], Geiselmann and Steinwandt suggested a distributed version of the basic design described in §3.3 through §3.5. This variant splits the monolithic circuit into multiple interconnected chips, each handling a submatrix. Performance is limited by inter-chip communication bandwidth (which is presently much lower than on-chip bandwidth), and thus decreased; however, the smaller chips are far easier to manufacture than wafer-scale circuits, thereby addressing the technological hurdles discussed in §3.2.

3.7 Parametrization

It remains to determine the optimal values of K and ρ, and derive concrete cost estimates. This involves implementation details and technological quirks, and obtaining precise figures appears rather hard. We thus derive expressions for the various cost measures, based on parameters which can characterize a wide range of implementations. We then substitute values that reasonably represent modern technology, and optimize for these.

3.7.1 Technology parameters

The technology parameters used in our estimates are as follows.

- Let $\mathcal{A}_t$, $\mathcal{A}_f$ and $\mathcal{A}_d$ be the average wafer area occupied by a logic transistor, an edge-triggered flip-flop and a DRAM bit, respectively (including the related wires).
- Let $\mathcal{A}_w$ be the area of a wafer.
- Let $\mathcal{A}_p$ be the wafer area occupied by an inter-wafer connection pad (see §3.2).
- Let $\mathcal{C}_w$ be the construction cost of a single wafer (in large quantities).
- Let $\mathcal{C}_d$ be the cost of a DRAM bit that is stored off the wafers (this is relevant only to the FPGA implementation; see below).
- Let $\mathcal{T}_d$ be the reciprocal of the memory DRAM access bandwidth of a single wafer (relevant only to FPGA).

- Let $\mathcal{T}_l$ be the time it takes for signals to propagate through a length of circuitry (averaged over logic, wires, etc.).
- Let $\mathcal{T}_p$ be the time it takes to transmit one bit through a wafer I/O pad.

We consider three concrete implementations: custom-produced "logic" wafers, custom-produced "DRAM" wafers (which reduce the size of DRAM cells at the expense of size and speed of logic transistors) and an FPGA-based design using off-the-shelf parts. Rough estimates of the respective parameters, circa 2003–2005, are given in Table 3.1.

The "Custom-130-L" set corresponds to logic-oriented 130μm feature size CMOS VLSI process on 30cm silicon wafers, exemplified by the Intel Pentium 4 "Northwood" processor; the figures are derived from the concrete parameters of this processor (see [131] for details), and are consistent with ITRS 2001 [91]. The "Custom-130-D" set likewise corresponds to DRAM-oriented 130μm process, and "Custom-90-D" corresponds to DRAM-oriented 90μm feature size VLSI process (cf. ITRS 2003 [93]). We assume a basic cost of \$5,000 per wafer.[13]

Custom VLSI construction entails an additional Non Recurring Engineering (NRE) cost, for one-time tasks such as creation of the lithographic masks; this cost is on the order of \$1M–\$10M, depending on the process's details and the circuit's complexity.

Our main focus is on custom-built hardware, in accordance with the focus on throughput cost. In practice, however, we are often concerned about solving a small number of factorization problems. In this case, it may be preferable to use off-the-shelf components (especially if they can be dismantled and reused, or if discreteness is desired). The "FPGA" parameters give an indication on the cost using off-the-shelf hardware, namely Field Programmable Gate Array (FPGA) chips connected in a two-dimensional grid, where each chip handles a block of mesh nodes. The numbers are derived for an Altera Stratix EP1S25F1020C7 FPGA augmented by external DRAM chips. For simplicity of calculation, the parameters are normalized so that one LE is considered to occupy 1 area unit, and one FPGA chip is considered a "wafer". See [131] for details.

3.7.2 Deriving the cost of the device

The estimated cost of the improved routing-based architecture, as of §3.5, is derived using the following assumptions and approximations:

- The number of mesh nodes is D/ρ.
- The values in $Q_j[\cdot]$ (i.e., the nonzero entries of A) can be stored in DRAM banks in the vicinity of the nodes, where (with an efficient representation) they occupy $h\rho\lg(D)\mathcal{A}_d$ per node.

[13] This is the basic cost of an untested wafer. The full costs depend on the architecture (e.g., the fault-tolerance capabilities) and are discussed separately.

	Custom-130-L (130nm logic process)	Custom-130-D (130nm DRAM process)	Custom-90-D (90nm DRAM process)	FPGA (Altera Stratix (+ DRAM)
$\mathcal{A}_t$	2.38 μm^2	2.80 μm^2	1.40 μm^2	0.05
$\mathcal{A}_f$	19.00 μm^2	22.40 μm^2	11.20 μm^2	1.00
$\mathcal{A}_d$	0.70 μm^2	0.20 μm^2	0.10 μm^2	$\emptyset$
$\mathcal{A}_p$	4 000 μm^2	4 000 μm^2		$\emptyset$
$\mathcal{A}_w$	$6.36 \cdot 10^{10}$ μm^2	$6.36 \cdot 10^{10}$ μm^2	$6.36 \cdot 10^{10}$ μm^2	25 660
$\mathcal{C}_w$	\$5,000	\$5,000	\$5,000	\$150
$\mathcal{C}_d$	$\emptyset$	$\emptyset$	$\emptyset$	$\$4 \cdot 10^{-8}$
$\mathcal{T}_d$	$\emptyset$	$\emptyset$	$\emptyset$	$1.1 \cdot 10^{-11}$ sec
$\mathcal{T}_l$	$1.46 \cdot 10^{-11}$ sec/μm	$1.80 \cdot 10^{-11}$ sec/μm		$1.43 \cdot 10^{-9}$ sec
$\mathcal{T}_p$	$4 \cdot 10^{-9}$ sec	$4 \cdot 10^{-9}$ sec		$2.5 \cdot 10^{-9}$ sec

Table 3.1: Implementation hardware parameters
Here, $\emptyset$ denotes values that are inapplicable and taken to be zero. Blank entries were not needed for our calculations.

- The $P_i[c]$ registers can be moved to DRAM banks, where they occupy $\rho K \mathcal{A}_d$ per node.
- The $P'_j[c]$ registers can also be moved to DRAM. However, to update the DRAM when a message is received we need temporary storage of $\lg(\rho) + K$ bits. Throughout the D/ρ steps of a routing operation, each node gets just 1 message on average (or less, due to annihilation). Thus, one $(\lg(\rho) + K)$-bit register per node suffices; if the register is still in use when another message arrives, that message can be kept unconsumed, allowing it to wonder around until the register is freed up. This occupies $\rho K \mathcal{A}_f$ per node when $\rho < 2$, and $\rho K \mathcal{A}_d + 2(\lg(\rho) + K)\mathcal{A}_f$ per node when $\rho \geq 2$.
- The bitwise logic related to the $P_i[c]$ registers, the $P'_i[c]$ and the last K bits of the $R[j]$ registers together occupy $20 \cdot \min(\rho,2) K \mathcal{A}_t$ per node.
- The $R[j]$ registers occupy $(\lg(D) + K)\mathcal{A}_f$ per node.
- The rest of the mesh circuitry (clock distribution, DRAM access, clockwise transposition routing, I/O handling, inner products, etc.) occupies $(200 + 30\lg(D))\mathcal{A}_t$ per node.
- Let $\mathcal{A}_n$ be total area of a mesh node, obtained by summing the above (we get different formulas for $\rho < 2$ vs. $\rho \geq 2$).
- Let $\mathcal{A}_m = A_n D/\rho$ be the total area of the mesh nodes (excluding inter-wafer connections).
- Let $\mathcal{N}_w$ be the number of wafers required to implement the matrix step, and let $\mathcal{N}_p$ be the number of inter-wafer connection pads per wafer. For single-wafer designs, $\mathcal{N}_w = 1/\lfloor \mathcal{A}_w/\mathcal{A}_m \rfloor$ and $\mathcal{N}_p = 0$. For multiple-wafer designs, these values are derived from equations for wafer area and bandwidth: $\mathcal{N}_w \mathcal{A}_w = \mathcal{A}_m + \mathcal{N}_w \mathcal{N}_p \mathcal{A}_p$, $\mathcal{N}_p = 4 \cdot 2 \cdot \sqrt{D/(\rho \mathcal{N}_w)} \cdot (\lg D + K) \cdot \mathcal{T}_p/(\sqrt{\mathcal{A}_n}\mathcal{T}_l)$.
- Let $\mathcal{N}_d$ be total number of DRAM bits (obtained by evaluating $\mathcal{A}_m$ for $\mathcal{A}_f = \mathcal{A}_t = 0, \mathcal{A}_d = 1$).

Algorithm	Technology	ρ	K	Wafers/ chips/ PCs	Construction cost $\mathcal{C}_s$	Run time $\mathcal{T}_s$ (sec)	Throughput cost $\mathcal{C}_s\mathcal{T}_s$ (US\$ × years)
Routing	Custom-130-L	0.51	107	19	\$94,600	960 (16 min)	2.87 (opt)
Routing	Custom-130-L	0.11	532	288	\$1,440,000	120 (2 min)	5.48
Routing	Custom-130-D	42.10	208	1	\$5,000	14 600 (4.1 hours)	2.32 (opt)
Routing	Custom-130-D	216.16	42	0.37	\$2,500	228 000 (2.6 days)	18.0
Routing	FPGA	5473.24	25	64	\$13,800	10 600 000 (123 days)	$4.66 \cdot 10^3$ (opt)
Routing	FPGA	243.35	60	2500	\$253,000	1 420 000 (17 days)	$1.14 \cdot 10^4$
Sorting	Custom-130-L		1	273	\$4,100,000	1 210 000 (14 days)	$1.57 \cdot 10^5$
Sorting	Custom-130-L		$\gg 1$	273	\$4,100,000	121 000 (34 hours)	$1.57 \cdot 10^4$
Serial	PCs		32	1	\$4,460	83 300 000 (3 years)	$1.18 \cdot 10^4$

Table 3.2: Cost of the matrix step for the throughput-optimized matrix

- Let $\mathcal{N}_a$ be the number of DRAM bit accesses (reads+writes) performed throughout the matrix step. We get: $\mathcal{N}_a = 2D(2hDK + Dh \lg D)$, where the first term due to the $P'_i[c]$ updates and the second term accounts for reading the matrix entries.
- Let $\mathcal{C}_s = \mathcal{N}_w \mathcal{C}_w + \mathcal{N}_d \mathcal{C}_d$ be the total construction cost for the matrix step.
- The full block Wiedemann algorithm consists of $2D/K$ matrix-by-vector multiplications, each of which consists of $h\rho$ routing operations, each of which consists of $2\sqrt{D/\rho}$ clocks. Each clock cycle takes $\mathcal{T}_l\sqrt{A_n}$.
  The time $\mathcal{T}_s$ taken by the full block Wiedemann algorithm is thus:
  $\mathcal{T}_s = 4D^{3/2}h\mathcal{T}_l\sqrt{\rho\mathcal{A}_n}/K + \mathcal{N}_a\mathcal{T}_d/\mathcal{N}_w$.

3.8 Cost estimates for 1024-bit composites

3.8.1 Cost estimates for the throughput-optimized matrix

Table 3.2 lists the estimated cost of the improved routing-based circuit of §3.5, for the throughput-optimized matrix (§5.4.1.2), using several choices of ρ and K. It also lists the cost of the sorting-based circuits (see §3.2) and a bandwidth-based lower bound on PC implementation (see §1.7.4). The lines marked by "(opt)" give the parameter choice that minimize the throughput cost for each type of hardware.

The third line describes a routing-based design whose throughput cost is roughly 67,000 times lower than that of the original sorting-based circuit (or 6,700 times lower than sorting with $K \gg 1$). Notably, this is a single-wafer device, which eliminates the technological problem of connecting multiple wafers with millions of parallel wires, as necessary in the original design of [22] (see §3.2). The fourth line shows that significant parallelism can be gained essentially for free: here, 88% of the wafer area is occupied simply by the DRAM banks needed to store the input matrix in a very compact representation, so further reduction in construction cost seems impossible.

Algorithm	Implementation	ρ	K	Wafers/ chips/ PCs	Construction cost $\mathcal{C}_s$	Run time $\mathcal{T}_s$ (sec)	Throughput cost $\mathcal{C}_s\mathcal{T}_s$ (US\$ $\times$ years)
Routing	Custom-130-L	0.51	136	6,030	\$30.1M	$3.3 \cdot 10^6$ (39 days)	$3.21 \cdot 10^6$ (opt)
Routing	Custom-130-L	0.11	663	9000	\$500.0M	$4.3 \cdot 10^5$ (4.9 days)	$6.08 \cdot 10^6$
Routing	Custom-130-D	41.12	306	391	\$2.0M	$4.6 \cdot 10^7$ (1.5 years)	$2.84 \cdot 10^6$ (opt)
Routing	Custom-130-D	261.55	52	120	\$0.6M	$1.0 \cdot 10^9$ (32 years)	$1.89 \cdot 10^7$
Routing	FPGA	17,757.70	99	13,567	\$3.5M	$2.3 \cdot 10^{10}$ (726 years)	$2.52 \cdot 10^9$ (opt)
Routing	FPGA	144.41	471	$6.6 \cdot 10^6$	\$1000.0M	$7.3 \cdot 10^8$ (24 years)	$2.40 \cdot 10^{10}$
Serial	PCs		32	1	\$1.3M	$5.7 \cdot 10^{12}$ (180 centuries)	$2.35 \cdot 10^{11}$

Table 3.3: Cost of the matrix step for the runtime-optimized matrix

ρ	K	m^2	IO lines	chips	chip size (mm^2)	run time (days)
1982	69	128^2	1280	308^2	36^2	238
1982	69	128^2	2560	308^2	36^2	121
1551	31	254^2	2048	100^2	50^2	481
1551	31	254^2	4096	100^2	50^2	255

Table 3.4: Cost of the matrix step for the runtime-optimized matrix using a distributed routing-based architecture using 130μm CMOS process.

3.8.2 Cost estimates for the runtime-optimized matrix

The runtime-optimized matrix, resulting from ordinary relation collection contains optimized for running time (see §5.4.1.2), has 250 times more columns than the throughput-optimized matrix: $D \approx 10^{10}$. We assume the same average column density, $h = 100$.

Using the formulas given in the previous section, we obtain the costs in Table 3.3 for the custom and FPGA implementations, for various parameter choices. The fourth line shows that here too, significant parallelism can be attained at very little cost (88% of the wafer area is occupied by DRAM storing the input). As can be seen, the improved mesh has a feasible cost also for the runtime-optimized matrix, and its cost is a small fraction of the cost of the alternatives, and of the relation collection step.

This matrix is too large to fit into a single monolithic circuit with present (or near-future) technology, even with wafer-scale integration; the storage alone, even using compact DRAM memory, already exceeds a wafer's capacity. The technological feasibility of this architecture thus depends critically on the ability to make high bandwidth inter-wafer connections. We can rely on emerging technological solutions as exemplified in §3.2), or invoke the distributed version of [75] mentioned in §3.6. Cost estimates for the latter are given in Table 3.4. This table lists the chip size and running time, as affected by inter-chip IO bandwidth (given as the number of number of full-speed, unidirectional I/O lines per chip) and the choice of architecture parameters.

These chip sizes are quite reasonable: similarly sized chips are commonly manufactured nowadays, and the design is highly regular and inherently fault-tolerant. The chip interconnect bandwidth is still rather high, and presently it is only marginally feasible.

We surmise that for 1024-bit composites, our routing-based architecture can feasibly carry out the linear algebra step at a practical cost — modulo the aforementioned technological scalability challenges. In particular, even for the conservative runtime-optimized matrix parameter settings, the predicted cost is significantly lower than that of the corresponding sieving step.

Chapter 4

A scalable pipelined architecture for the NFS linear algebra step

4.1 Overview

This chapter continues the investigation the NFS linear algebra step. While in the previous chapter we have shown that the NFS linear algebra step for large composites can be implemented at a practical cost, which is below that of sieving, the resulting designs have a major drawback: they require (at least one of) extremely large chips, non-local wiring or high-bandwidth chip interconnects, and thus pose significant technological hurdles.

We proceed to describe a new systolic design for the NFS linear algebra step, and specifically for the matrix-by-vector multiplications which dominate the cost of the Wiedemann algorithm. In its simplest form, it consists of a one dimensional chain of identical chips with purely local interconnects, which (from a practical standpoint) makes it an attractive alternative to previous wafer-scale mesh proposals. For higher efficiency it can be generalized to a two-dimensional array of chips. Unlike previous proposals, this device has standard chip sizes (which are cheap and widely available), purely local interconnects, and can use standard DRAM chips for some of its components. In addition, the new design is highly scalable: there is no need to commit to particular problem sizes and densities during the chip design phase, and there is no need to limit the problem size to what can be handled by a single wafer. Since a single chip design of small fixed size can handle a wide range of sparse matrix problems (some of which may be related to the solution of partial differential equations rather than cryptography), the new architecture can have additional applications, greatly reduced technological uncertainties, and lower initial NRE cost.

Unlike previous routing based proposals, whose complex data flows required simulation of the whole device and were not provably correct, the present device has a simple and deterministic data flow, so that each unit can be simulated independently. This facilitates the simulation and actual construction of meaningful proof-of-concept sub-devices.

For a conservative choice of 1024-bit NFS matrix parameters, the concrete cost estimate of the design is 0.4M US$×year, an order of magnitude lower than the best previous proposal (including that of Chapter 3 and variants thereof).

The present design adapts efficiently and naturally to operations over any finite field GF(q), since it does not depend on the in-transit pairwise cancellation of values in GF(2).

As part of our design but also of independent interest, we describe a new error-detection scheme adaptable to any implementation of Wiedemann's algorithm (such as that of the preceding section). Under reasonable assumptions, the new scheme can be used to detect computational errors with probability arbitrarily close to 1 and at negligible cost.

4.2 The architecture

In the following, the task and notation are as given in §1.7. As in the previous chapters, the architecture we describe is general, but for the sake of clarity we specify a concrete instance where various design parameters are chosen suitably to 1024-bit RSA factorization. These concrete parameters are designated by angular brackets (e.g., $D \langle\!\langle = 10^{10} \rangle\!\rangle$). §4.4 provides additional details about the parameters and §4.5 discusses the cost of the device for these parameters.

As before, we shall unravel the architecture in several stages, where each stage generalizes its predecessor and (when appropriately parametrized) improves its efficiency.

4.2.1 A basic scheme

The proposed hardware device is preloaded with a compressed representation of the sparse matrix $A \in \mathrm{GF}(q)^{D \times D}$,[1] as will be detailed below. For each of the multiplication chains described in §1.7.1, we load the input vector $\vec{v}$ and iteratively operate the device to compute the vectors $A\vec{v}, A^2\vec{v}, \ldots, A^{D/K}\vec{v}$.

We begin by describing an inefficient and highly simplified version of the device, to illustrate its high-level data flow.[2] This simplified device consists of $D \langle\!\langle = 10^{10} \rangle\!\rangle$ *stations* connected in a pipeline. The i-th station is in charge of the i-th matrix row, and contains a list of the $\langle\!\langle \approx 100 \rangle\!\rangle$ non-zero entries in that row. It is also in charge of the i-th entry of the output vector, and contains a corresponding accumulator $W'[i]$.

In each multiplication, the input vector $\vec{v} \in \mathrm{GF}(2)^D$ is fed into the top of the pipeline, and moves down as in a shift register. As the entries of $\vec{v}$ pass by, the i-th station looks at all vector entries v_j passing through it, identifies the ones corresponding to the non-zero matrix entries $A_{i,j}$ in row

[1] We describe the operation over an arbitrary finite field GF(q), for applicability to the solution of linear systems over any such field. In the context of the NFS factoring algorithm, $q = 2$.

[2] This basic version can be thought of as the linear-algebra-step analogue of the pipeline-of-adders variant of TWINKLE (see §2.2.1). Many of the improvements described in the following have corresponding analogues in the TWIRL architecture of Chapter 2.

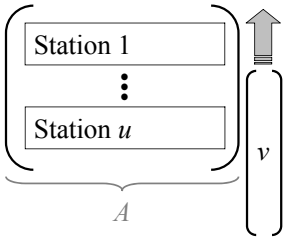

Figure 4.1: Distributing the entries of A onto stations

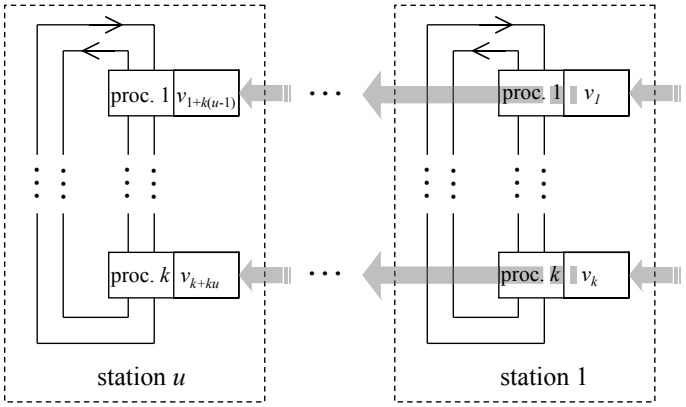

Figure 4.2: Subdivision of a chip into stations and processors

i, and for those entries adds $A_{i,j} \cdot v_j$ to its accumulator $W'[i]$. Once the input vector has passed all stations in the pipeline, the accumulators $W'[\cdot]$ contain the entries of the product vector $A\vec{v}$. With additional shift operations, these can now be sequentially off-loaded and fed back to the top of the pipeline in order to compute the next multiplication.

The one-dimensional chain of stations can be split across several chips: each chip contains one or more complete stations, and the connections between stations may span chip boundary. Note that since communication is unidirectional, inter-chip I/O latency is not a concern (though we do need sufficient bandwidth; the amount of bandwidth needed will increase in the variants given below, and is taken into account in the cost analysis of Section 4.5).

4.2.2 Compressed row handling

Since the matrix A is extremely sparse, it is wasteful to dedicate a complete station for handling each row of A, as it will be idle most of the time. Thus, we partition A into $u \langle\!\langle = 9600 \rangle\!\rangle$ horizontal stripes and assign each such stripe to a single station (see Figure 4.1). The number of rows per station is $\mu \approx D/u \langle\!\langle = 2^{20} \rangle\!\rangle$, and each station contains μ accumulators $W'[i]$ with i ranging over the set of row indices handled by the station.

Each station stores all the non-zero matrix entries in its stripe, and contains an accumulator for each row in the stripe. As before, the input vector $\vec{v}$ passes through all stations, but now there are just u of these (rather than D). Since the entries of $\vec{v}$ arrive one by one, each station implicitly handles a $\mu \times D$ submatrix of A at each clock cycle. Each station now needs to be capable of handling multiple events simultaneously; this is described below.

4.2.3 Compressed vector transmission

For additional efficiency, we add parallelism to the vector transmission. Instead of each station processing a single entry of $\vec{v}$ in each clock-cycle, we process $\vec{v}$ in chunks of k $\langle\!\langle = 32 \rangle\!\rangle$ consecutive entries.[3] The inter-station pipeline is thickened by a factor of k. The vector $\vec{v}$ now passes in chunks of k entries over an inter-station pipeline (in Figure 4.2 from right to left); in each clock cycle, each station obtains such a chunk from the previous station (to its right), processes it and passes it to the next station (to its left). The first (rightmost) station gets a new part of the vector received from the outside. At each clock cycle, each station now implicitly handles a $\mu \times k$ submatrix of A.

Each station is comprised of k processors, each connected to a separate pipeline line (see Figure 4.2), and these k processors inside each station are connected via γ $\langle\!\langle = 2 \rangle\!\rangle$ *intra-station channels*, which are circular shift registers spanning the station. The μ accumulators $W'[i]$ contained in this station are partitioned equally between the k processors.

For processing a k-element chunk of the vector, each of the k processors has to decide whether the vector element v_i it currently holds is relevant for the station it belongs to, i.e., whether any of the μ matrix rows handled by this station contains a non-zero entry in column i. If so, then v_i should be communicated to the processor handling the corresponding accumulator(s) and handled there. This is discussed in the following subsection.

4.2.4 Processing vector elements

Fetching vector elements. The relevance of a vector entry v_i to a given station depends only on i, which is uniquely determined by the clock cycle and the processor (out of the k) it reached. Consequently, each processor needs to read the content of one pipeline line (to which it is attached) at a predetermined set of clock cycles, specific to that processor, which is constant across multiplications and easily precomputed. We encode it as follows; this is essentially an alternative representation of the non-zero matrix entries belonging to the processor, in terms of events instead of indices.

Each processor contains a *fetches table* which instructs it when to read the next vector element from the pipeline. It contains *fetch events*, represented as triplets (τ, f, ℓ) where τ is an δ_{u} $\langle\!\langle = 7 \rangle\!\rangle$-bit integer, f is a one-bit flag and ℓ is a $\lceil \lg(\gamma) \rceil$-bit integer. Such a triplet means: "ignore the incoming vector entries for τ clock cycles; then, if $f = 1$, read the input vector element and transmit it on the ℓ-th intra-station channel".[4] The table is read sequentially, and is stored in compact DRAM-type memory.

Updating the accumulators. Once a relevant vector element v_i has been fetched by some processor and copied to an intra-station channel, we still need to handle it by adding $A_{j,i} \cdot v_i$

[3] The choice of k depends mainly on the number of available I/O pins for inter-chip communication.

[4] The flag f is used to handle the cases where the interval between subsequent fetches is more than $2^{\delta_{\mathrm{u}}} - 1$.

to the accumulator $W'[j]$, for every row j handled by this station for which $A_{j,i} \neq 0$. These accumulators (usually just one) may reside in any processor in this station. Thus, each processor also needs to occasionally fetch values from the intra-station channels and process it. Similarly to above, the timing of this operation is predetermined, identical across multiplications, easily precomputed, and compactly stored.

To this end, each processor also holds an *updates table* containing *update events* represented as a 5-tuple (τ, f, ℓ, j', x) where τ is an $\delta_{\mathrm{f}}\langle\!\langle = 7 \rangle\!\rangle$-bit integer, f is a one-bit flag, ℓ is a $\lceil \lg(i) \rceil$-bit integer, j' is a $\lceil \lg(\mu/k) \rceil$-bit integer and x is a field element.[5] Such a 5-tuple means: "ignore the intra-station channels for τ clock cycles; then, if $f = 1$, read the element $y \in \mathrm{GF}(q)$ currently on channel ℓ, multiply it by x, and add the product to the j'-th accumulator in this processor". This table is also read sequentially and stored in compact DRAM-type memory.

During a multiplication, each processor essentially just keeps pointers into those two tables (which can actually be interleaved in a single DRAM bank), and sequentially executes the events described therein.

An update operation requires a multiplication over $\mathrm{GF}(q)$ and addition of the product to an accumulator stored in DRAM (which is very compact but has high latency). These operations occur at non-regular intervals, as prescribed by the updates table; the processors use small queues to handle congestion, where a processor gets several update events within a short interval. Crucially, the load on these queues is known in advance as a side effect of computing the tables. If some processor is over-utilized or under-utilized, we can change the assignments of rows to stations, or permute the matrix columns, to even the load.

Handling dense rows. All the entries arriving from the intra-station channels while the updated vector is stored into the DRAM have to be held in the processor's queues. As the random-access latency of DRAM is quite large ($\approx$ 70ns), the entries must not arrive too fast. Some of the rows of A are too dense, and could cause congestions of the queues and intra-station channels. To overcome this problem we split such dense rows into several sparser rows, whose sum equals the original (which can be thus computed at the end of the multiplication). In this way we also ensure that all stations have a similar load and handle the same number of rows. This increases the matrix size by an insignificant amount ($\langle\!\langle \approx 10^6 \rangle\!\rangle$ additional rows[6] added to the original $D\,\langle\!\langle \approx 10^{10} \rangle\!\rangle$), and the post-processing required to re-combine the split rows is trivial. Further load-balancing can be done using a scattered matrix representation [34].

Precomputation and simulation. The content of the two tables used by each processor fully encodes the matrix entries. These tables are precomputed once for each matrix A, e.g., using ordinary PCs. Once computed, they allow us to easily simulate the operation of any processor at any clock cycle, as it is completely independent of the rest of the device and of the values of the input vectors. We can also accurately (though inefficiently) simulate the whole device. Unlike the mesh-based approach of the previous chapter, we do not have to rely on heuristic run time assumptions for the time needed to complete a single matrix-vector multiplication.

[5] Over GF(2), $x = 1$ always and can thus be omitted.

[6] Extrapolated from a preprocessed RSA-155 NFS matrix from [44], provided to us by Herman te Riele.

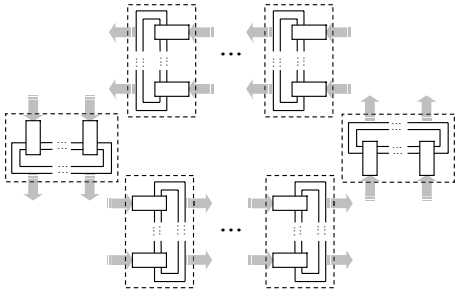

Figure 4.3: Arranging the stations into a circle

4.2.5 Skewed assignment for iterated multiplication

In the above scheme, once we have started feeding the initial vector $\vec{v}$ into the pipeline, after $(D/k) + u$ clock cycles[7] the vector $\vec{v}$ has passed through the complete pipeline and the vector $A \cdot \vec{v}$ is fully computed but spread throughout the stations. More precisely, each of the u stations contains $\mu = D/u$ consecutive components of $\vec{v}$, and we next want to compute the matrix-by-vector product $A \cdot A\vec{v}$. Thus, we need to somehow feed the computed result $A\vec{v}$ back into the inter-station pipeline.

To feed the vector $A\vec{v}$ back into the inter-station pipeline, first we physically close the station interconnects into a circle as depicted in Figure 4.3; this can be done by appropriate wiring of the chips on the PCB. We also place a memory bank of D/u GF(q) elements at each of the u stations. Collectively, denote these banks by W; their role will be related to that of the $W'[i]$ accumulators, which we shall henceforth denote collectively by W'. At the beginning of each multiplication chain, the initial vector $\vec{v}$ is loaded into W sequentially, station by station.

During a multiplication, the content of W is rotated, by having each station treat its portion of W as a FIFO of k-tuples: in each clock cycle it sends the last k-tuple of its portion of W to the next station, and accepts a new k-tuple from the previous station. Meanwhile, the processors inside each station function exactly as before, by tapping the flow of k-tuples of vector elements in W at some fixed point (e.g., the head of the FIFO in that station). Thus, after D/k clock cycles, we have completed a full rotation of the content of W and the multiplication result is ready in the accumulators W. A key point here is that each station sees the contents of W in a cyclic order but starting at a different offset; due to the commutativity of addition in GF(q) this does not affect the final result.

Having obtained the matrix-by-vector product, we can now continue to the next multiplication simply by switching the roles (or equivalently, the contents) of the memory banks W and accumulators W': this amounts to a simple local operation in each processor (note that the size and distribution among processors of the cells $W[\cdot]$ and the cells $W'[\cdot]$ is indeed identical). Thus, the matrix-by-vector multiplications can be completed at a rate of one per D/k cycles.

[7] Actually slightly more, due to the need to empty the station channels and processor queues.

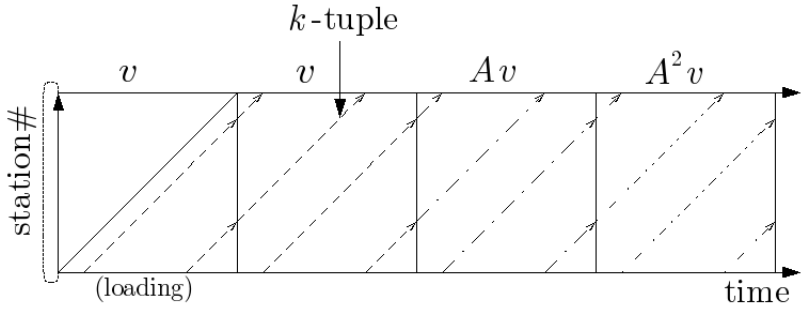

Figure 4.4: Movement of vector element k-tuples through the circle of stations.

This temporal packing of multiplications can be illustrated by plotting the place vs. time of vector elements during the operation of the device, as depicted in Figure 4.4.

4.2.6 Amortizing matrix storage cost

Recall that in the block Wiedemann algorithm, we actually execute $2K$ multiplication chains with different initial vectors but identical matrix A. These are separated into two phases, and in each phase we can handle these K chains in parallel. An important observation is that we can handle these K chains using a single copy of the matrix (whose representation, in the form of the two event tables, has so far dominated the cost). This greatly reduces the amortized circuit cost per multiplication chain, and thus the overall cost per unit of throughput.

The above is achieved simply by replacing every field element in W and W' by a K-tuple of field elements, and replacing all field additions and multiplications with element-wise operations on the corresponding K-tuples. The event tables and the logic remain the same. Note that the input and output of each station (i.e., the pipeline width) is now bigger, containing $k \cdot K$ field elements.

4.2.7 Two-dimensional chip array

As described above, each of the processors inside the station incorporates two types of memory storage: a fixed storage for the representation of the matrix elements (i.e., the event tables), and vector-specific storage (W and W') which increases with the parallelization factor K. Ideally, we would like to use a large K in order to reduce the amortized cost of matrix storage. However, this is constrained by the chip area available for W and W'.

To obtain further parallelization without increasing the chip sizes, we could simply run several copies of the device in parallel. By itself, this does not improve the cost per unit of throughput. But now all of these devices use identical storage for the matrix representation, and access it sequentially at the same rate, so in fact we can "feed" all of them from a single matrix representation. In this variant, the event tables are stored in an external DRAM bank, and are connected

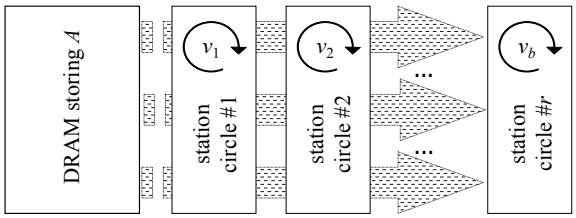

Figure 4.5: Using external memory to store the matrix A, and b parallel devices (each hosting a circle of stations)

to the chips hosting the processors and chain-specific storage through a unidirectional pipeline, as illustrated in Figure 4.5. Note that communication remains purely local—there are no long broadcast wires.

This variant splits each of the monolithic chips used by the previous variants into a standard DRAM memory chip for matrix storage, plus a chain of small ASIC chips for the processors and the storage of the vectors. By connecting $b \langle\!\langle = 90 \rangle\!\rangle$ such ASIC chips to each DRAM chip, we can increase the blocking factor K by a factor of b without incurring the cost of duplicate matrix storage.

4.3 Fault detection and correction

4.3.1 Importance

To successfully complete the Wiedemann algorithm, the device must compute all the matrix-by-vector multiplications without a single error. For the problem parameters of interest the multiplications will be realized by tens of thousands of chips operating over several months, and it would be unrealistic to hope (or alternatively, expensive to ensure) that all the computations will be faultless; even a single bit-flip error occurring during these months, if gone undetected, could render the results of this extensive computation effectively useless.[8] The same concern arises for other special-purpose hardware designs, and also for software implementations on commodity hardware. It is thus crucial to devise algorithmic means for detecting and correcting faults. In the following we describe a method for detecting such errors soon after they occur, with arbitrarily high probability, and at a very low cost in terms of complication, device size and running time.

4.3.2 A generic scheme

As above we use the notation of §1.7, except that we generalize to a matrix A and vector $\vec{v}$ over any finite field GF(q). For integer factoring via NFS, $q = 2$.

[8] Note the contrast with the NFS sieving step, which can tolerate both false positive and false negative errors in its smoothness tests.

A simple real time error-detection scheme would be to apply a linear test, as follows. During a preprocessing stage, choose a random $d \times D$ matrix B for an appropriate d, precompute on a reliable host computer and store in the hardware the $d \times D$ matrix $C = BA$, and verify that $B\vec{w}' = C\vec{w}$ whenever the hardware computes a new product $w' = A\vec{w}$ for some $\vec{w}$. Over GF(q) each row of the matrix B reduces the probability of an undetected error by a factor of q, and thus for $q = 2$ we need at least a hundred rows in B to make this probability negligible.

Since each one of the dense $100 \times D$ matrices B and C contains about the same number of 1's as the sparse $D \times D$ matrix A (with one hundred 1's per row), this linear test can triple the storage and processing requirements of the hardware, and meshes poorly with the overall design whose efficiency relies heavily on the sparseness of the matrix rows. Note that we cannot solve this problem by making the $100 \times D$ matrix B sparse, since this would greatly reduce the probability of detecting single bit errors.

In the following we describe an alternative error-detection scheme, which provides (effectively) an arbitrarily small error probability at a negligible cost, under reasonable assumptions. It inspects only the computed (possibly erroneous) matrix-by-vector products, and can thus be applied to any implementation of Wiedemann's algorithm.

Detection. Let $\vec{w}_0, \vec{w}_1, \vec{w}_2, \ldots \in \mathrm{GF}(q)^D$ denote the sequence of vectors computed by the device, where $\vec{w}_0 = \vec{v}$. To verify that indeed $\vec{w}_i = A^i\vec{v}$ for all $i > 0$, we employ the following randomized linear test. For a small integer d ⟪$= 200$⟫, choose a single vector $\vec{b} \in \mathrm{GF}(q)^D$ uniformly at random, and precompute on a reliable computer the single vector

$$\vec{c} \stackrel{\text{def}}{=} \left(\vec{b}A^d\right)^{\mathrm{T}} \quad .$$

After each $\vec{w}_i$ is computed, compute also the inner products $vectrb\vec{w}_i$ and $\vec{c}^{\,\mathrm{T}}\vec{w}_i$ (which are just field elements). Save the last d results of the latter in a small shift register, and after each multiplication test the following condition:

$$\vec{b}^{\,\mathrm{T}}\vec{w}_i = \vec{c}^{\,\mathrm{T}}\vec{w}_{i-d} \quad . \tag{4.1}$$

If equality does not hold, declare that at least one of the last d multiplications was faulty.

Correctness. If no faults have occurred then (4.1) holds since both sides equal $\vec{b}^{\,\mathrm{T}}A^i\vec{v}$. Conversely, we will argue that the first faulty multiplication $\vec{w}_j \neq A\vec{w}_{j-1}$ will be detected within d steps with overwhelming probability, under reasonable assumptions.

Let us first demonstrate this claim in the simplest case of a single transient error $\vec{\varepsilon}$ which occurs in step j. This changes the correct vector $A^j\vec{v}$ into the incorrect vector $\vec{w}_j = A^j\vec{v} + \vec{\varepsilon}$. All the previous $\vec{w}_i$ for $i < j$ are assumed to be correct, and all the later $\vec{w}_i$ for $i > j$ are assumed to be computed correctly, but starting with the incorrect $\vec{w}_j$ in step j. It is easy to verify that the difference between the correct and incorrect values of the computed vectors $\vec{w}_i$ for $i > j$ evolves as $A^{i-j}\vec{\varepsilon}$, and due to the effective randomness of the matrix A (generated by the sieving step) these error vectors are likely to point in random directions in the D-dimensional space GF(q)^D.

The device has d chances to catch the error by considering pairs of computed vectors which are d apart, with the first vector being correct and the second vector being incorrect. The probability that all these d random error vectors will be orthogonal to the single random test vector $\vec{b}$ is expected to be about q^{-d}, which is negligible; the computational cost was just two vector inner products per matrix-by-vector multiplication.

The analysis becomes a bit more involved when we assume that the hardware starts to malfunction at step j, and adds (related or independent) fault patterns to the computed result after the computation of each matrix-vector product from step j onwards. Let the result of the i-th multiplication be $\vec{w}_i = A\vec{w}_{i-1} + \vec{\varepsilon}_i$, where the vector $\vec{\varepsilon}_i$ is the error in the output of this multiplication. We consider the first fault, so $\vec{\varepsilon}_i = 0$ for all $i < j$. Assume that $j \geq d$ ($j < d$ will be addressed below). By the linearity of the multiplication and the minimality of j, we can expand the above recurrence to obtain $\vec{w}_i = A^i\vec{v} + \sum_{i'=j}^{i}(A^{i-i'}\vec{\varepsilon}_{i'})$ $(i \geq j)$. Plugging this into (4.1) and canceling out the common term $\vec{b}^{\mathrm{T}} A^i \vec{v}$, we get that for $j \leq i < j + d$, (4.1) is equivalent to:

$$\vec{b}^{\mathrm{T}} r_i = 0 \qquad \text{where} \qquad r_i = \sum\nolimits_{i'=j}^{i}(A^{i-i'}\vec{\varepsilon}_{i'}) \ . \tag{4.2}$$

We assume that each error $\vec{\varepsilon}_i$ is one of at most $(qD)^\alpha$ possibilities for some $\alpha \ll D/d$ (e.g., $\langle\!\langle \alpha = 10^5 \rangle\!\rangle$), regardless of A and $\vec{b}$. This suffices to enumerate all reasonably likely combinations of local faults (corrupted matrix entries, faulty pipeline connections, errors in $\mathrm{GF}(q)$ multipliers, memory bit flips, etc.). We also make the simplifying (though not formally correct) assumption that $A^{10}, ..., A^{d-1}$ are random matrices drawn uniformly and independently.[9] Then for any fixed values of $\vec{\varepsilon}_i$, the vectors in the set $R = \{r_i\}_{i=j+10}^{j+d-1}$ are drawn uniformly and independently from $\mathrm{GF}(q)^D$ (recall that $\vec{\varepsilon}_j \neq 0$), and thus the probability that the span of R has dimension less than the maximal $d - 10$ is smaller than $dq^{-(D-d)}$ (which is a trivial upper bound on the probability that one of the d random vectors falls into the span of the others). By assumption, there are at most $(qD)^{\alpha d}$ possible choices of $(\vec{\varepsilon}_i)_{i=j+1}^{j+d}$. Hence, by the union bound, the probability that the span of R has dimension less than $d - 10$ is at most $(qD)^{\alpha d} \cdot dq^{-(D-d)} = d \cdot q^{\alpha d \log_q D + d - D}$, which is negligible. Conditioned on the span of R having full rank $d - 10$, the probability of the random vector $\vec{b}$ being orthogonal to the span of R is $q^{-(d-10)}$, which is also negligible. Hence, with overwhelming probability, at least one of the tests (4.2) for $j + 10 < i < j + d$ will catch the fault in $\vec{w}_j$.

Startup and finalization. Note that the test (4.1) applies only to $i > d$, and moreover that our analysis assumes that the first d multiplications are correct. Thus, for each of the $2K$ multiplication chains of block Wiedemann, we start the computation by computing the first d

[9] The sieving and preprocessing steps of NFS yield a matrix A that has nearly full rank and is"random-looking" except for some biases in the distribution of its values: A is sparse (with density $\langle\!\langle \approx 100/10^{10} \rangle\!\rangle$) and its density is decreasing with the row number. The first few self-multiplications increase the density exponentially and smoothen the distribution of values, so that A^{10} has full and uniform density. The independence approximation is applicable since we are looking at simple local properties (corresponding to sparse error vectors), which are "mixed" well by the matrix multiplication. While the resulting matrices do have some hidden structure, realistic fault patterns are oblivious to that structure.

multiplications on a reliable general-purpose computer (possibly redundantly for verification), and then load the state (including the queue of $\vec{c}^{\mathrm{T}}\vec{w}_i$ values for $i = 0, \ldots, d$) into the device for further multiplications.

Also note that in the analysis, the results of the j-th multiplications are implicitly checked by (4.1) for $i = j, \ldots, j + d - 1$. Thus, in order to properly check the last d multiplications in each chain, we run the device for d extra steps and discard the resulting vectors but still test (4.1).

Recovery. The above method will detect a fault within d clock cycles (with overwhelming probability), but will not correct it. Once the fault is detected, we must backtrack to a known-good state without undoing too much work. Assuming a sufficiently low probability of error, it is simplest to dump a full copy of the current vector $\vec{w}_i$ from the device into a general-purpose computer, at regular but generously-spaced intervals; this can be done by another special station tapping the pipeline. The backup vectors may be stored on magnetic media, and thus their storage has negligible cost. When a fault is detected, the faulty component can be replaced (or a spare device substituted) and the computation restarted from the last known-good backup.

4.3.3 Device-specific considerations

Implementation. The above scheme requires only the computation of two inner products ($\vec{b}^{\mathrm{T}}\vec{w}_i$ and $\vec{c}^{\mathrm{T}}\vec{w}_i$) for each multiplication. In the proposed hardware device, this is achieved by one additional station along the pipeline, which taps the vector entries flowing along the pipeline and verifies their correctness by the above scheme. This station contains the entries of $\vec{b}$ and $\vec{c}$ in sequential-access DRAM. For each of the K vectors being handled, the station processes a k-tuple of vector entries at every clock cycle, keeps the d most recent values of $\vec{c}^{\mathrm{T}}\vec{w}_i$ in a local FIFO queue at this station, and performs the test according to (4.1).

Halving the cost. The storage cost can be halved by choosing $\vec{b}$ pseudorandomly instead of purely randomly; the number of multipliers can also be nearly halved by choosing $\vec{b}$ to be very sparse.

Using faulty chips. In addition to the above high-level error-recovery scheme, it is also useful to work around local faults in the component chips: this increases chip yield and prevents the need to disassemble multi-chip devices if a fault was discovered after assembly. To this end, the proposed device offers a significant level of fault tolerance due to its uniform pipelined design: we can add a "bypass" switch to each station, which effectively removes it from the pipeline (apart for some latency). Once we have mapped the faults, we can work around any fault in the internals of some station (this includes the majority circuit area) by activating the bypass for that station and assigning its role to one of a few spare stations added in advance. The chip containing a local fault then remains usable, and only slightly less efficient.

4.4 Parametrization

4.4.1 Matrix parameters

For 1024-bit composites we shall assume that the input size matches the runtime-optimized matrix parameter set from §5.4.1.2, i.e., a matrix of size $D \times D$ for $D \approx 10^{10}$ with a density of 100 entries per column. This matrix size, which was also evaluated in previous sections and others' follow-up works, represents a conservative estimate with generous margins; in particular, it is smaller than the matrix that is expected to be produced by TWIRL.

4.4.2 Technology parameters

We shall assume 90nm chip manufacturing technology with DRAM-type process, i.e., Custom-90-D setting of §3.7.1, with a net chip area of 1 cm^2, a per-chip I/O bandwidth of 1024 Gbit/s, and a clock rate of 1GHz. A DRAM access is assumed to take 70 clock cycles. These parameters are quite realistic, if not conservative, with current technology.

4.5 Cost estimates

4.5.1 Cost for 1024-bit NFS matrix step

Clearly there are many possibilities for fixing the different parameters of our device, depending on such parameters as desired chip size and number of chips. It is also possible to combine the above design with the distributed version of [75] (see §3.6), thereby giving up the homogeneity and purely local communication but decreasing the dimension of the handled vectors. In the following, we consider a specific parameter set, which focuses on practicality using VLSI technology circa 2005. Further details are given below.

We employ a 300×90 array of ASIC chips. Each column of 300 chips contains $u = 9600$ stations (32 per chip). Each station consists of $k = 32$ processors, communicating over $\gamma = 2$ intra-station channels, with a parallelization factor of 10. Each of the 300 rows, of 90 chips each, is fed by a 108Gbit DRAM module.[10] Overall, the blocking factor is $K = 10 \cdot 90 = 900$. This array can complete all multiplication chains in ≈ 2.4 months.

The total chip area, including the matrix storage, is less than 90 full 30cm wafers. Assuming a silicon cost of US$ 5000 per wafer, and a factor 4 increase for overheads such as faulty chips, packaging, testing and assembly, the total construction cost is under US$ 2M. The throughpost cost is thus under 0.4M US$×year. This is significantly below that of sieving with comparable technology (see Chapter 2).

[10] Note that the fault tolerance method from §4.3 protects the memory as well.

4.5.2 Further details

To derive concrete cost and performance estimates for the 1024-bit case, several implementation choices for parameters, such as δ_u, δ_f, γ, τ, have been determined experimentally as follows. For the above problem and technology parameters, and a large randomly drawn matrix, we used a software simulation of a station to check for congestions in bus and memory accesses, and chose design parameters for which such congestions never occur experimentally. Recall that the device's operation is deterministic and repetitive (see §4.2.4), so the simulation accurately reflects the device's operation with the given parameters.

In the following we briefly mention some aspects of the circuit area and its analysis, as used to derive the overall estimate above. Note that we employ the split design of Section 4.2.7, which puts the matrix storage in plain DRAM chips and the logic and vector storage in ASIC chips. For these parameters, memory storage dominates area: approximately 97% of the ASIC chip area is occupied by the DRAM which stores the intermediate vectors (i.e., W and W'). Thus, as in previous chapters, the suitable chip production process is a DRAM process optimized for maximum memory density (at the expense of slightly larger logic circuits); hence our choice of the Custom-90-D setting.

Each of the $k \langle\!\langle = 32 \rangle\!\rangle$ processors in each of the 32 stations in each of the 300×90 ASIC chips contains the following logic:

- A K/b-bit register for storing the K/b-tuples of GF(2) elements flowing along from the inter-station pipeline ($\approx 8 \cdot K/b$ transistors).
- A K/b-bit register for each of the $\gamma \langle\!\langle = 2 \rangle\!\rangle$ intra-station channels ($\approx 8 \cdot \gamma \cdot K/b$ transistors).
- A FIFO queue $\langle\!\langle$of depth 2$\rangle\!\rangle$ for storing elements arriving on the inter-station pipeline along with the number of the internal bus onto which the respective element is to be written. For this $\approx 2 \cdot 8 \cdot (K + \lceil \lg(\gamma) \rceil)$ transistors per queue entry are sufficient.
- A FIFO queue $\langle\!\langle$of depth 4$\rangle\!\rangle$ for storing elements arriving on the intra-station channels that have to be XORed to the vector. Each entry consists of a K/b-tuple of bits for the vector and a row number in the submatrix handled by the station has to be stored. This occupies $\approx 4 \cdot 8 \cdot (K/b + \lceil \lg \lceil D/(ku) \rceil \rceil) \langle\!\langle = 4 \cdot 8 \cdot (10 + 15) \rangle\!\rangle$ transistors per queue entry.

In addition to the registers and queues, we need some logic for counters (to identify the end of a vector and to decide when to read another element from a bus), multiplexers, etc. For the parameters of interest, under 1500 transistors are sufficient for this; the 32×32 processors on each chip occupy about 3.2mm^2.

The required DRAM storage consists of three parts:

- For storing $2 \cdot K/b$ vectors in $\text{GF}(2)^{\lceil D/(uk) \rceil}$,: $2 \cdot K/b \cdot D/(uk)$ bit $\langle\!\langle \approx 650$ Kbit$\rangle\!\rangle$.
- For the fetches table: $\delta_u + 1 + \lceil \lg(\gamma) \rceil$ bits per entry.

- For the updates table: $\delta_f + 1 + \lceil \lg(\gamma) \rceil + \lceil \lg \lceil D/(uk) \rceil \rceil$ bits per entry.

Overall, the DRAM on each chip occupies $\langle\!\langle \approx 67\text{mm}^2 \rangle\!\rangle$. The number of clock cycles for each of the $\approx 2D/K$ matrix-by-vector multiplications is D/k (plus a small overhead, $\ll 1000$ cycles, for emptying queues and internal buses).

4.5.3 Comparison to previous designs

The optimized mesh-based architecture considered in Chapter 3 ([70]), adapted to 90nm technology and distributed onto using 85×85 chips of size 12.25 cm^2 each following [76] (to approach the constraints of present technology), would require about 17.7 months to process the above matrix, and the higher complexity of that design entails a lower clock rate in that design. Comparing throughput per silicon area, the new device is over 10 times more efficient; moreover it has much smaller individual chips and no need for non-local wiring.

This concludes our description and evaluation of the new NFS hardware architectures.
A summary of our results and their implications is presented in Chapter 6.

Chapter 5

Analysis of NFS parameters

5.1 Overview

The results of the preceding chapters rely heavily on numerical estimates of various parameters in the Number Field Sieve, for problem sizes that are yet to be realized experimentally. However, the NFS algorithm is notoriously hard to analyze and predict; even its asymptotic behavior is understood only up to a exponent of $1 + o(1)$, and relies on plausible yet unproven conjectures. Some aspects are understood heuristically and can be approximated numerically (e.g., relation yields and the optimal choice of factor bases), while for other aspects little is known beyond a handful of smaller-scale experimental results (e.g., cycle yields and the effect of large primes). Previous works (e.g., [197, 35, 130]) relied on rough methods for extrapolation from past experiments. These do not suffice for our purpose, and indeed, as shall be shown here, turn out to be misleading when stretched to 1024-bit composites.

To facilitate and support the results of the preceding chapters, it is necessary to tackle this uncertainty. The corresponding analysis is described in this chapter. We have employed all known techniques to obtain to most plausible and detailed estimates of NFS parameters for 1024-bit and 768-bit composites. Moreover, our analysis extends the known numerical estimation techniques in order to derive a wealth of additional information on intermediate values, such as the frequency of candidate relations, at various stages of the algorithm. Considering the NFS polynomials, a crucial component of the NFS algorithm, we demonstrate polynomials that are significantly superior to those obtained by the commonly used method.

Beyond the necessity of these estimates for deriving concrete costs, they also played a crucial role in guiding the optimization and fine-tuning of various trade-offs inherent in the TWIRL architecture, thereby significantly contributing to the cost reduction. Additionally, the extended analysis facilitated upper-bounding the cost of several auxiliary steps in the NFS algorithms (e.g., cofactor factorization) which might have, by themselves, dominated the total cost after the drastic cost reduction in the main steps. This bolsters the significance of the new results as reflecting the total cost of NFS.

Organization. In §5.2, we review and extend the three main techniques used for estimating parameters and costs: straightforward extrapolation from asymptotic behavior, semi-numerical evaluation by use of asymptotic smoothness probability functions, and experimental evaluation by direct smoothness testing. These methods are, in this order, increasingly accurate but also increasingly complicated, and each is applicable to a different subset of the NFS parameters.

In §5.3, we present carefully selected NFS polynomials that were used for our experiments; these are necessary for reproducing the results and are of use for other research pursuing the same goal.

We then employ these techniques in two ways. First, in §5.4, we derive the main NFS parameters for large composites (1024 and 768 bits) using the asymptotic approach, and evaluate the result via the other two approaches. The results are instructive, in pointing out the dangers of relying on the asymptotic extrapolations alone (a common practice in regard to NFS). Then, in §5.5, we investigate a different parameter setting for the sieving parameters, guided by a more refined analysis using smoothness probability functions. This setting is used for the TWIRL device of §2.2, and we consider it in depth in that context.

5.2 NFS parameter estimation techniques

We begin by describing the methods employed to estimate the yield of the NFS. The basic techniques are well-known, but at several places we extend them to improve accuracy, obtain additional information, and make them applicable to a more general setting.

5.2.1 Notes on the Number Field Sieve

The following provides some additional pertinent background about the Number Field Sieve, beyond the basic description in §1.5.

5.2.1.1 Cycle yield

As explained in §1.5.3, the number of relations required to obtain $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})$ independent cycles (when using NFS with large primes) is determined by the matching behavior of the large primes, and this behavior varies from factorization to factorization and is not yet well understood.[1] Furthermore, the behavior gets considerably more complicated if 2 or 3 large primes are allowed in the rational and algebraic norms, which is beneficial and customary in current factorizations (e.g., [44][127]). The uncertainty about the matching behavior of the large primes is the main reason that it is currently impossible to give reliable estimates for the difficulty of factoring numbers that are much larger than the numbers we have experience with. For that reason, we shall investigate several parameter choices which assume different cycle yields.

[1] Obviously, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})$ distinct non-free full relations suffice, but this is necessary only in the yet-to-be-observed worst-case situation where large primes cannot be paired at all. Likewise, for any number of permissible number large primes ($\ell_{\mathbf{a}}$ and $\ell_{\mathbf{r}}$), $\pi(V_{\mathbf{r}}) + \pi(V_{\mathbf{a}})$ distinct relations always suffice.

5.2.1.2 Sieving effort

In the standard variant of NFS, for smoothness bounds $U_{\mathbf{r}}$,$U_{\mathbf{a}}$ and a sieving region of size S, the run time of the sieving effort is dominated by the number of times the progressions P_i induced by the factor base hit the sieving region. This value is approximately proportional to[2]

$$W \stackrel{\text{def}}{=} S \cdot (\log\log(U_{\mathbf{r}}) + \log\log(U_{\mathbf{a}})) \ .$$

5.2.1.3 Coppersmith's multi-polynomial version

As shown by Coppersmith [47], an improvement of the regular NFS can be obtained by replacing the single degree-d polynomial f with a set F containing multiple irreducible degree-d polynomials with a shared root m modulo n. In that case, a relation is a pair of coprime integers (a, b) with $b > 0$ such that $N_{\mathbf{r}}(a, b)$ is $(U_{\mathbf{r}}, V_{\mathbf{r}}, \ell_{\mathbf{r}})$-semismooth and $b^d f(a/b)$ is $(U_{\mathbf{a}}, V_{\mathbf{a}}, \ell_{\mathbf{a}})$-semismooth for some $f \in F$. The goal is to find $\pi(U_{\mathbf{r}}) + |F|(\pi(U_{\mathbf{a}}) - \pi(\min\{U_{\mathbf{r}}, U_{\mathbf{a}}\})/d!)$ cycles. First, sieving is used to find a set $\tilde{\mathcal{S}} \subset \mathcal{S}$ of sieve locations $(a, b) \in \tilde{\mathcal{S}}$ with $(U_{\mathbf{r}}, V_{\mathbf{r}}, \ell_{\mathbf{r}})$-semismooth rational norms $N_{\mathbf{r}}(a, b)$ with a and b coprime. Next, a smoothness test different from sieving is used to test $b^d f(a/b)$ for $(U_{\mathbf{a}}, V_{\mathbf{a}}, \ell_{\mathbf{a}})$-semismoothness for all $(a, b) \in \tilde{\mathcal{S}}$ and all $f \in F$; in [47], the Elliptic Curve Method (ECM) is suggested. The approximate run time of the relation collection becomes proportional to

$$S \cdot \log\log(U_{\mathbf{r}}) + \Upsilon \cdot |\tilde{\mathcal{S}}| \cdot |F|$$

where Υ is a constant that determines the cost ratio between ECM and sieving. This constant depends on the (semi-)smoothness bounds $U_{\mathbf{a}}, V_{\mathbf{a}}, \ell_{\mathbf{a}}$ employed, and on many practical factors such as the cost ratio between logic and storage.

5.2.1.4 Asymptotic NFS costs and parameters

We now briefly survey the asymptotic behavior of the Number Field Sieve.

Following standard notation (see [129]), let $L_x[r, \alpha]$ denote any function of x that equals

$$e^{(\alpha + o(1))(\ln x)^r (\ln\ln x)^{1-r}} \quad \text{for } x \to \infty \ , \tag{5.1}$$

where α and r are real numbers with $0 \le r \le 1$. Under appropriate assumptions[3], optimization of the parameters for minimum run time (on a serial computer) for $n \to \infty$, leads to the asymptotic costs and major parameters given in the first line of Table 5.1.[4]

Coppersmith's multi-polynomial variant of NFS (see §5.2.1.3) runs, asymptotically, slightly faster; see the second line of Table 5.1.[5] The third line shows the effect when we optimize for minimal

[2] This disregards significant memory scalability and caching effects.

[3] Namely, a special case of Assumption 3.

[4] See our paper [131] for the detailed derivations.

[5] For Coppersmith's variant the exact expressions are too long to fit in the table; see [131].

NFS variant	Optimized for	Number of operations	Throughput cost	Bounds $U_{\mathbf{r}}$, $U_{\mathbf{a}}$; matrix size D	Sieve region dimensions A, B
Ordinary	#ops	$(64/9)^{1/3}$	$(64/9)^{1/3}(2/3)$	$(8/9)^{1/3}$	$(8/9)^{1/3}$
Coppersmith	#ops	$1.90188361\ldots$	$2.85282541\ldots$	$0.95094180\ldots$	$0.95094180\ldots$
$\hookrightarrow$ "free matrix"	#ops	$1.86893284\ldots$	$3.14851348\ldots$	$1.04950449\ldots$	$0.93446642\ldots$
Circuits	throughput	$(5/3)^{4/3}$	$(5/3)^{4/3}$	$(5/3)^{1/3}(2/3)$	$(5/3)^{1/3}(5/6)$

Table 5.1: Asymptotic behavior of several NFS variants. An entry α means that the asymptotic value of this parameters is $L_n[1/3, \alpha]$.

sieving effort, in term of number of operations, and pretend that the matrix step is free. The fourth line shows the asymptotic effect of using relation collection without sieving §1.6.7 combined with one of the special-purpose linear-algebra devices of Chapter 3 or Chapter 4, with parameters chosen to minimize throughput cost as in [22].

These expressions provide some insight into parameter selection, but the presence of the $o(1)$ limits their practical value. The choice of polynomial f (hence, the correction factor ξ in $u_{\mathbf{a}}$), as well as the use of large primes, are believed to affect these values only by a constant factor (which disappears in the $o(1)$).

5.2.2 Extrapolation from asymptotics

5.2.2.1 Technique

The simplest approach to estimating the cost and parameters of NFS for large integers is to start with some representative factoring experiment of a smaller integer, and use the expected asymptotic cost scaling to perform extrapolation.[6] It proceeds as follows.

Let $\mathcal{R}$ indicate a resource required for a factorization effort. For instance, $\mathcal{R}$ could indicate the computing time or throughput cost (see §1.4), or it could be the factor base size, or the total matrix weight, or any other aspect of the factorization for which one measures the cost or size.

For a resource $\mathcal{R}$, let $C_{\mathcal{R}}(x)$ be a cost function that measures, asymptotically for $x \to \infty$ and in the relevant unit, how much of $\mathcal{R}$ is needed to factor composites $n' \approx x$ using NFS. For several resources a heuristic expression for this function is known. For example, when $\mathcal{R}$ measures the total minimized expected run time, then $C_{\mathcal{R}}(x) = L_x[1/3, (64/9)^{1/3}]$ as given in §5.2.1.4.

Assume that for the NFS factorization of some RSA modulus n', $\mathcal{R}_{n'}$ units of some resource $\mathcal{R}$ are known to be required (or sufficient). Then $\frac{C_{\mathcal{R}}(n)}{C_{\mathcal{R}}(n')}\mathcal{R}_{n'}$ is used as an estimate estimate how much of $\mathcal{R}$ would be required (resp., sufficient) for the factorization of RSA modulus n. The factorization of the 512-bit RSA challenge composite $n' =$ RSA-155 (see §1.3.1) is often used as the basis for such extrapolation.[7]

[6] This approach was used, for example, in [197, 35, 130].

[7] Larger composites have factored (see §1.3.1), but the necessary details were not published.

Similarly, suppose we consider two different parametrization settings for NFS (e.g., optimized for different goals), with corresponding cost functions $C^{\mathrm{I}}_{\mathcal{R}}(x)$, $C^{\mathrm{II}}_{\mathcal{R}}(x)$ for some resource $\mathcal{R}$. Then if the NFS factorization of some RSA modulus n requires $\mathcal{R}^{\mathrm{I}}_n$ units of $\mathcal{R}$ in the first setting, we may extrapolate that second setting would require $\mathcal{R}^{\mathrm{II}}_n = \frac{C^{\mathrm{II}}_{\mathcal{R}}(n)}{C^{\mathrm{I}}_{\mathcal{R}}(n)}\mathcal{R}_n$ units.

5.2.2.2 Caveats

In this type of estimate it is customary to ignore the $o(1)$ factors in the exponent when $C_{\mathcal{R}}$ is of the form (5.1). Based on frequent observations this is not unreasonable if $\log(n')$ and $\log(n)$ are close. For large scale extrapolations, however, omitting the $o(1)$'s may be an over-simplification that might produce misleading results.

Furthermore, even if $\log(n')$ and $\log(n)$ are close, $C_{\mathcal{R}}$-based extrapolation for resources $\mathcal{R}$ that are well understood in theory, may lead to results of dubious practical value, for the following reason.

Consider the RSA-155 [44] 512-bit factorization. By extrapolation, one would recommend a factor base size that is about 2.5 times larger than for a 462-bit factorization (such as RSA-140 [201]). In practice, however, the entire concept of factor base size is obscured by the use of multiple large primes and special q's: it turned out that using the same factor base (due to implementation limitations) did not severe degrade performance. This effect, where far-from-optimal factor base sizes still lead to only slightly suboptimal performance, is due to the flat behavior around the minimum of the run time curve as a function of the factor base size.[8]

However, run time increases sharply if the factor base size gets much too small (see [196]). This causes a potential danger for factor base size extrapolation which disregard the $o(1)$ term: a suboptimal small choice, in the region where the curve is relatively well-behaved for the factor base for n', may be extrapolated to a factor base size for n in the steep region of the curve for n, thereby leading to a much larger total run time for n than anticipated. This effect will be clearly observed, for one concrete case of interest, in §5.4.2.3.

5.2.3 Semi-smoothness probabilities

A finer approach for estimating the yield of relations and candidates in the Number Field Sieve, described in the next section, relies on asymptotic smoothness and semismoothness probabilities. To this end, we recall and define several number-theoretical functions. The study of such functions is at times termed *psixyology*; we shall henceforth occasionally refer to these as *psixyological functions*.[9]

[8] That is, the derivative of the cost as a function of the factor base size is small near the minimum cost, as one may expect.

[9] The term "psixyology" was coined by Moree [144] after a function $\psi(x, y)$ that is central to the analysis.

Asymptotic probability	Predicate (where $U = x^{1/u}$, $V = x^{1/v}$, $W = x^{1/w}$)	Term
$\rho(u)$	$\Pi_{\rho(u),x}(z) \stackrel{\text{def}}{=} z_{(1)} < U$	U-smooth
$\sigma(u,v)$	$\Pi_{\sigma(u,v),x}(z) \stackrel{\text{def}}{=} z_{(1)} \le V \ \wedge \ z_{(2)} \le U$	"semismooth" in [14]
$\sigma_\ell(u,v)$	$\Pi_{\sigma_\ell(u,v),x}(z) \stackrel{\text{def}}{=} U < z_{(\ell)} \le V \ \wedge \ z_{(\ell+1)} \le U$	strictly (U,V,ℓ)-semismooth
$\bar{\sigma}_2(u,v,w)$	$\Pi_{\bar{\sigma}_2(u,v,w),x}(z) \stackrel{\text{def}}{=} U \le z_{(2)} \le V \ \wedge \ z_{(3)} \le U \ \wedge z_{(1)}z_{(2)} \le W$	restricted strictly semismooth
$\sum_{\ell'=0}^{\ell} \sigma_{\ell'}(u,v)$	$z_{(\ell)} \le V \ \wedge \ z_{(\ell+1)} \le U$	(U,V,ℓ)-semismooth

Table 5.2: Summary of smoothness functions, conditions, notation and terminology

5.2.3.1 Definitions

For a non-negative integer ℓ and positive reals u, v such that $0 < v < u < 1$, let $\sigma_\ell(u,v)$ denote the probability, for $x \to \infty$, that a random positive integer smaller than x is strictly $(x^{1/u}, x^{1/v}, \ell)$-semismooth (see §1.5.2). In particular, $\sigma_0(u,u)$ is the probability of $x^{1/u}$-smoothness and equals the Dickman $\rho(u)$ function $u^{-u+o(1)}$ for $u \to \infty$.[10] This generalizes the $\sigma(u,v)$ function of Bach and Peralta [14] (whose $\sigma(u,v)$ equals $\sigma_1(u,v) + \rho(u)$ in our notation) and the $\sigma_2(u,v)$ function of Lambert [113].

We define a restricted notion of strict $(u, v, 2)$-semismoothness, where the product of the two large primes is also bounded. For u, v, w such that $0 < w < v < u < 1$, let $\bar{\sigma}_2(u,v,w)$ be the probability, for $x \to \infty$, that a random integer $0 < z < x$ fulfills $x^{1/u} \le z_{(2)} \le x^{1/v}$, $z_{(3)} \le x^{1/u}$ and also $z_{(1)}z_{(2)} \le x^{1/w}$. This is a generalization of strict semismoothness, since $\sigma_2(u,v) = \bar{\sigma}_2(u,v,v/2)$.

These conditions and asymptotic functions are summarized in Table 5.2 and illustrated in Figure 5.1. For each asymptotic probability function φ, the table defines a corresponding predicate $\Pi_{\varphi,x}$ such that, for fixed parameters u, v, w, we have $\varphi = \lim_{x\to\infty} \Pr_z[\Pi_{\varphi,x}(z)]$ for a uniformly drawn integer $0 < |z| < x$.

5.2.3.2 Computation

Closed expressions for σ_ℓ and $\bar{\sigma}_2$ are not known. For ρ and σ_1, we used the numerical approximation methods of Bach and Peralta [14]. For σ_2, Lambert provided a derivation in [113, Section 4.3] and an effective numerical calculation method in [113, Section 4.4 and Appendix A]. We extend this to $\bar{\sigma}_2$ as follows:

Lemma 2. *Let u, v, w be such that $0 < w < v < u < 1/2$. Let $\alpha = 1/u$, $\gamma = 1/w$ and $\beta = \min(1/v, \gamma - \alpha)$. If $\beta \le \alpha$ then $\bar{\sigma}_2(u,v,w) = 0$, otherwise:*

$$\bar{\sigma}_2(u,v,w) = \frac{1}{2}\int_\alpha^\beta \left(\int_\alpha^{\min(\beta,\gamma-\lambda_1)} \frac{\rho(1-\lambda_1-\lambda_2)}{\alpha} \cdot \frac{d\lambda_2}{\lambda_2} \right) \frac{d\lambda_1}{\lambda_1}$$

[10] See Crandall and Pomerance [51], Canfield et al. [40] and de Bruijn [53].

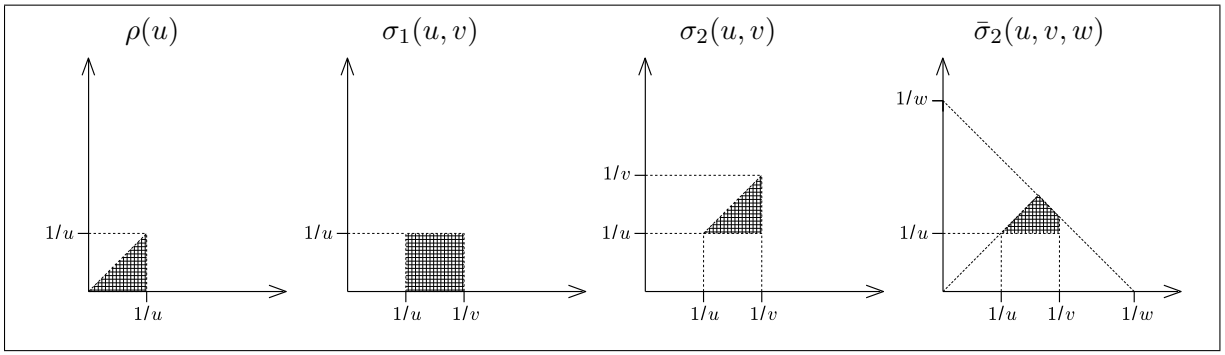

Figure 5.1: Illustration of the main (semi-)smoothness probability functions. In each case, for a given $x > 0$, an integer z whose two largest prime factors are $z_{(1)} \geq z_{(2)}$ (possibly 1) is represented by the point $(\log z_{(1)}/\log x, \log z_{(2)}/\log x)$. For each (semi-)smoothness function φ with fixed parameters, the shaded region represents the integers z fulfilling the (semi-)smoothness predicate $\Pi_{\varphi,x}$, and φ equals the measure of the shaded region (over uniformly drawn z s.t. $|z| \leq x$ and $x \to \infty$).

Proof outline. The non-trivial case $\beta > \alpha$ proceeds analogously to Lambert's derivation of σ_2. Define a function $\bar{\Psi}_2$ such that $\bar{\sigma}_2(u,v,w) = \lim_{x\to\infty} \bar{\Psi}_2(x, x^{1/v}, x^{1/u}, x^{1/w})$:

$$\bar{\Psi}_2(x, x^\beta, x^\alpha, x^\gamma) \stackrel{\text{def}}{=} \sum_{x^\alpha \leq p_1 \leq x^\beta} \sum_{x^\alpha \leq p_2 \leq \min\{p, x^\gamma/p_1\}} \left|\left\{z \leq x : z_{(1)} = p,\ z_{(2)} = p_2,\ z_{(3)} \leq x^\alpha\right\}\right|$$

This is identical to the function $\Psi_2(x, x^\beta, x^\alpha)$ of [113] except for the limits of the summation. We then follow the proof of [113, Theorem 4.3.1], substituting $\bar{\Psi}_2$ for Ψ_2 and adjusting the limits of summation and integration accordingly. In the transition to the analogue of equation [113, Eq. (4.3)], note that the two-dimensional set over which we integrate remains symmetric. The claim follows. □

5.2.3.3 Implementation

For the yield estimates of §5.4.2 and §5.5, we wrote programs that compute all of the aforementioned asymptotic smoothness functions. Our primary implementation is in C++, using a generalization of the effective technique of [113, Section 4.4] to $\bar{\sigma}_2$. This implementation also solves critical numerical stability problems which caused infinite loops in [113, Appendix A]. It also employs caching of common expression and look-up techniques for improved performance, in order to execute the methods of the next section in acceptable time.

For reference, we also wrote two additional implementations employing direct numeric integration (following [14] and [113, Section 4.3] instead of symbolic manipulation of the integrals. One was written in C++ using the *GNU Scientific Library* [79], and the other in Mathematica [219].

The consistency checking facilitated by multiple implementations helped us identify and correct a previously unnoticed error in the derivation and code of [113].[11] Where applicable, consistency

[11] On Page 97 there, a factor of 1/2 due to the Jacobian is omitted, leading to incorrect results for σ_2 [114].

with the tables in [14] and the code in [113] was also verified.

5.2.4 Estimates via smoothness probabilities

The psixyological smoothness probability functions defined in the previous section let us express and estimate the yield of full and partial relations in the Number Field Sieve.

5.2.4.1 Assumptions

First, we make the standard heuristic assumption that the values assumed by $N_{\mathbf{r}}(a,b)$ and $N_{\mathbf{a}}(a,b)$ behave like independent random integers of similar size with respect to their (semi-)smoothness probability, apart from the correction factor ξ defined as follows.

For integer z, let $\eta(U,z)$ denote the largest U-smooth factor of z. For random integers[12] $z \gg U$, the expected value $\Gamma(U)$ of $\ln \eta(U,z)$ is known to be

$$\Gamma(U) \stackrel{\text{def}}{=} \sum_{p<U,\, p \text{ prime}} (\log p)/(p-1) \ .$$

Conversely, for a given NFS polynomial f, the expected value $\Gamma_f(U)$ of $\ln \eta(U, N_{\mathbf{a}}(a,b))$ taken over $(a,b) \in \mathcal{S}$ can be estimated experimentally by averaging $\ln \eta(U, N_{\mathbf{a}}(a,b))$ over a large random set of coprime pairs (a,b). The correction factor that measures f's advantage is then defined as the difference between the $\ln \eta(\cdot)$ values assumed by the norms vs. the random case:[13]

$$\xi \stackrel{\text{def}}{=} \exp(\, \Gamma_f(2^{30}) - \Gamma(2^{30})\,) \ .$$

Typically we have $\xi > 1$ because f is chosen so that $N_{\mathbf{a}}(a,b)$ tends to have unusually many small factors (e.g., see [146]). The constant 2^{30} is chosen to be much larger than the small factors typically considered in the choice of f, and is otherwise inconsequential.

We also assume that asymptotic (semi-)smoothness probabilities, as captured by $\rho(u)$, $\sigma_\ell(u,v)$ and $\bar{\sigma}_2(u,v,w)$, for fixed ℓ, u, v and w, serve as a good estimate for concrete smoothness probabilities for the numbers of the magnitude arising in the Number Field Sieve (see, e.g., [14],[146],[113]).

The combinations of these assumption is semi-formally captured by the following:

Assumption 3. *Let $\varphi_{\mathbf{r}}, \varphi_{\mathbf{a}} \in \{\rho(u), \sigma_\ell(u,v), \bar{\sigma}_2(u,v,w)\}$ for any fixed u, v, w, ℓ (which may differ between $\varphi_{\mathbf{r}}$ and $\varphi_{\mathbf{a}}$). Let $\Pi_{\varphi_{\mathbf{r}},x}$ and $\Pi_{\varphi_{\mathbf{a}},x}$ be the corresponding predicates from Table 5.2. Let $x_{\mathbf{r}}, x_{\mathbf{a}} \gg 1$. Let (a,b) be drawn uniformly from $\mathcal{S}$ conditioned on $N_{\mathbf{r}}(a,b) \approx x_{\mathbf{r}}$ and $N_{\mathbf{a}}(a,b)/\xi \approx y_{\mathbf{r}}$. Then:*

$$\Pr[\, \Pi_{\varphi_{\mathbf{r}},x_{\mathbf{r}}}(N_{\mathbf{r}}(a,b)) \wedge \Pi_{\varphi_{\mathbf{a}},x_{\mathbf{a}}}(N_{\mathbf{a}}(a,b))\,] \ \approx \ \varphi_{\mathbf{r}} \cdot \varphi_{\mathbf{a}} \ .$$

We have experimentally tested this assumption for several parameter choices; see §5.2.5 and §5.4.3.

[12] In an asymptotic sense, when drawn uniformly from an increasingly large range.

[13] Our ξ fulfills a role similar to $e^{\alpha(f)}$ in [146].

5.2.4.2 Estimating relations yield

Define:

$$u_{\mathbf{r}} = \frac{\log(N_{\mathbf{r}}(a,b))}{\log(U_{\mathbf{a}})} \quad , \quad u_{\mathbf{a}} = \frac{\log(N_{\mathbf{a}}(a,b)/\xi)}{\log(U_{\mathbf{a}})}$$
$$v_{\mathbf{r}} = \frac{\log(N_{\mathbf{r}}(a,b))}{\log(V_{\mathbf{r}})} \quad , \quad v_{\mathbf{a}} = \frac{\log(N_{\mathbf{a}}(a,b)/\xi)}{\log(V_{\mathbf{a}})} \quad .$$

By the above assumption, for integers $\ell'_{\mathbf{r}}, \ell'_{\mathbf{a}} \geq 0$, a sieve location (a,b) randomly drawn from $\mathcal{S}$ forms an $(\ell'_{\mathbf{r}}, \ell'_{\mathbf{a}})$-partial relation with probability

$$\sigma_{\ell'_{\mathbf{r}}}(u_{\mathbf{r}}, v_{\mathbf{r}}) \cdot \sigma_{\ell'_{\mathbf{a}}}(u_{\mathbf{a}}, v_{\mathbf{a}}) \quad .$$

conditioned on the magnitude of $N_{\mathbf{r}}(a,b)$ and $N_{\mathbf{a}}(a,b)$. Integration of these probabilities over the sieving region gives an estimate for the total yield of $(\ell'_{\mathbf{r}}, \ell'_{\mathbf{a}})$-partial relations. A correction factor $6/\pi^2 \approx 0.608$ is applied to all results to account for the probability that randomly drawn a and b are coprime, since other pairs (a,b) are discarded during NFS sieving.

In the case of Coppersmith's variant (see §5.2.1.3), an estimate of $|\tilde{\mathcal{S}}|$ is obtained by integrating the $\sigma_{\ell'_{\mathbf{r}}}(u_{\mathbf{r}}, v_{\mathbf{r}})$ values over the sieving region.

5.2.4.3 Estimating candidates yield

The above standard method lets us estimate the yield of full and partial relations, given the NFS polynomial and sieving region. However, the full relation collection step involves multiple stages of filtering, each testing for specific semi-smoothness properties. Sieving devices like TWIRL address only certain tests, leaving numerous candidates which undergo futher filtering in later stages. It is crucial to verify that the number of such initial candidates is sufficiently small so that subsequent filtering does not become a bottleneck. Moreover, in the cascaded sieves variant of TWIRL (see §2.3.5), the algebraic-side sieve handles only the pairs (a,b) that passed the rational sieve, and it should be verified that the latter are sufficiently infrequent; this is crucial for achieving the high parallelism factor of $s_A = 32{,}768$ inspected pairs per clock cycle.

We extend the method of the previous section to estimate these quantities. That is, we estimate the number of candidates at the relevant points in the algorithm by writing down the appropriate smoothness probability, integrating it over the sieving region and (analogously to §5.2.4) multiplying the result by the correction factor $6/\pi^2$.

For integer z, let $\mu(U,z) \stackrel{\text{def}}{=} z/\eta(y,z)$ denote the largest non U-smooth factor of z. In effect, the sieving step identifies the pairs (a,b) for which $\mu(U_{\mathbf{r}}, N_{\mathbf{r}}(a,b)) \leq V_{\mathbf{r}}^{\ell_{\mathbf{r}}}$ and $\mu(U_{\mathbf{a}}, N_{\mathbf{a}}(a,b)) \leq V_{\mathbf{a}}^{\ell_{\mathbf{a}}}$. Indeed, for $\ell_{\mathbf{a}}, \ell_{\mathbf{r}} \geq 2$, not all such pairs form relations.

Here we assume the common setting $\ell_{\mathbf{r}} = \ell_{\mathbf{a}} = 2$.[14] For brevity, let k_1 and k_2 denote the two largest prime factors of $N_{\mathbf{r}}(a,b)$; similarly, let κ_1 and κ_2 denote the two largest prime factors of

[14] That is, we assume two large primes per side; see e.g., [44] and [127]. The technique is easily generalized.

$N_{\mathbf{a}}(a,b)$:

$$k_1 \stackrel{\text{def}}{=} N_{\mathbf{r}}(a,b)_{(1)} \quad , \quad k_2 \stackrel{\text{def}}{=} N_{\mathbf{r}}(a,b)_{(2)} \quad , \quad \kappa_1 \stackrel{\text{def}}{=} N_{\mathbf{a}}(a,b)_{(1)} \quad , \quad \kappa_2 \stackrel{\text{def}}{=} N_{\mathbf{a}}(a,b)_{(2)} \quad .$$

The relevant types of candidates are as follows.[15]

Pass Rational Sieve (PRS): The pairs that pass the rational sieve are those that fulfill $\mu(U_{\mathbf{r}}, N_{\mathbf{r}}(a,b)) \leq V_{\mathbf{r}}^2$. Assuming $V_{\mathbf{r}}^2 < U_{\mathbf{r}}^3$, the above is equivalent to the following: $(k_1, k_2 < U_{\mathbf{r}}) \vee (U_{\mathbf{r}} < k_1 \leq V_{\mathbf{r}}^2 \wedge k_2 < U_{\mathbf{r}}) \vee (U_{\mathbf{r}} < k_1, k_2 \wedge k_1 k_2 \leq V_{\mathbf{r}}^2)$. Accordingly, the probability that (a,b) fulfills this can be estimated by $\rho(u_{\mathbf{r}}) + \sigma_1(u_{\mathbf{r}}, v_{\mathbf{r}}/2) + \bar{\sigma}_2(u_{\mathbf{r}}, v_{\mathbf{r}}/2, v_{\mathbf{r}}/2)$.

Pass Both Sieves (PBS): the probability that a pair (a,b) passes both sieves is obtained by multiplying the above by the analogous expression for the algebraic side: $(\rho(u_{\mathbf{r}}) + \sigma_1(u_{\mathbf{r}}, v_{\mathbf{r}}/2) + \bar{\sigma}_2(u_{\mathbf{r}}, v_{\mathbf{r}}/2, v_{\mathbf{r}}/2)) \cdot (\rho(u_{\mathbf{a}}) + \sigma_1(u_{\mathbf{a}}, v_{\mathbf{a}}/2) + \bar{\sigma}_2(u_{\mathbf{a}}, v_{\mathbf{a}}/2, v_{\mathbf{a}}/2))$.

Pass Primality Testing (PPT): For pairs that passed both sieves, the smooth factors are divided out to obtain $\mu(U_{\mathbf{r}}, N_{\mathbf{r}}(a,b))$ and $\mu(U_{\mathbf{a}}, N_{\mathbf{a}}(a,b))$.[16] If $\mu(U_{\mathbf{r}}, N_{\mathbf{r}}(a,b))$ is prime and larger than $V_{\mathbf{r}}$, or $\mu(U_{\mathbf{r}}, N_{\mathbf{a}}(a,b))$ is prime and larger than $V_{\mathbf{a}}$, then the pair is discarded. A pair (a,b) reaches and survives this test iff $(k_1, k_2 < U_{\mathbf{r}}) \vee (U_{\mathbf{r}} < k_1 \leq V_{\mathbf{r}} \wedge k_2 < U_{\mathbf{r}}) \vee (U_{\mathbf{r}} < k_1, k_2 \wedge k_1 k_2 \leq V_{\mathbf{r}}^2)$ and analogously for the algebraic side. The probability that this holds is estimated by $(\rho(u_{\mathbf{r}}) + \sigma_1(u_{\mathbf{r}}, v_{\mathbf{r}}) + \bar{\sigma}_2(u_{\mathbf{r}}, v_{\mathbf{r}}/2, v_{\mathbf{r}}/2)) \cdot (\rho(u_{\mathbf{a}}) + \sigma_1(u_{\mathbf{a}}, v_{\mathbf{a}}) + \bar{\sigma}_2(u_{\mathbf{a}}, v_{\mathbf{a}}/2, v_{\mathbf{a}}/2))$.

Rational Cofactor Factorizations (RCF): For pairs that survived primality testing, if the cofactor $\mu(U_{\mathbf{r}}, N_{\mathbf{r}}(a,b))$ is composite then it needs to be factored and tested for $V_{\mathbf{r}}$-smoothness. The size of the cofactor to be factored is bounded by $V_{\mathbf{r}}^2$. This step is reached and the factorization is performed if $(U_{\mathbf{r}} < k_1, k_2 \wedge k_1 k_2 \leq V_{\mathbf{r}}^2)$ and $(\kappa_1, \kappa_2 < U_{\mathbf{a}}) \vee (U_{\mathbf{a}} < \kappa_1 \leq V_{\mathbf{a}} \wedge \kappa_2 < U_{\mathbf{a}}) \vee (U_{\mathbf{a}} < \kappa_1, \kappa_2 \wedge \kappa_1 \kappa_2 \leq V_{\mathbf{a}}^2)$. The probability that this holds is estimated by $\bar{\sigma}_2(u_{\mathbf{r}}, v_{\mathbf{r}}/2, v_{\mathbf{r}}/2) \cdot (\rho(u_{\mathbf{a}}) + \sigma_1(u_{\mathbf{a}}, v_{\mathbf{a}}) + \bar{\sigma}_2(u_{\mathbf{a}}, v_{\mathbf{a}}/2, v_{\mathbf{a}}/2))$.

Rational Semi-Smooth (RSS): A pair reaches the rational cofactor factorization step and passes (or skips) it if indeed $N_{\mathbf{r}}(a,b)$ is $(U_{\mathbf{r}}, V_{\mathbf{r}}, \ell_{\mathbf{r}})$-smooth and (a,b) passed the algebraic sieve. For this to happen, the condition on the rational side is $(k_1, k_2 < U_{\mathbf{r}}) \vee (U_{\mathbf{r}} < k_1 \leq V_{\mathbf{r}} \wedge k_2 < U_{\mathbf{r}}) \vee (U_{\mathbf{r}} < k_1, k_2 \leq V_{\mathbf{r}})$, and the condition on the algebraic side is as in the previous step. Thus the probability is estimated by $(\rho(u_{\mathbf{r}}) + \sigma_1(u_{\mathbf{r}}, v_{\mathbf{r}}) + \sigma_2(u_{\mathbf{r}}, v_{\mathbf{r}})) \cdot (\rho(u_{\mathbf{a}}) + \sigma_1(u_{\mathbf{a}}, v_{\mathbf{a}}) + \bar{\sigma}_2(u_{\mathbf{a}}, v_{\mathbf{a}}/2, v_{\mathbf{a}}/2))$.

Algebraic Cofactor Factorizations (ACF): For pairs that are semismooth on the rational side, if the cofactor $\mu(U_{\mathbf{a}}, N_{\mathbf{a}}(a,b))$ is composite then it needs to be factored and tested for $V_{\mathbf{a}}$-smoothness. This step is reached and the factorization is performed iff $(U_{\mathbf{a}} <$

[15] This describes one plausible and commonly employed ordering of the filtering steps. Other variations are possible (e.g., performing the algebraic cofactor factorization before the rational cofactor factorization, or even before the rational primality testing), and indeed Kleinjung [106] has recently presented a method for interleaving the steps in an optimal way; our technique easily extends to these cases.

[16] Efficiency-wise, most prime factors smaller than $U_{\mathbf{r}}$ or $U_{\mathbf{a}}$ are reported by TWIRL; see §2.3.7.2.

$\kappa_1, \kappa_2 \wedge \kappa_1\kappa_2 \leq V_{\mathbf{a}}^2$) and also the rational-side condition of the previous step holds. The corresponding probability is estimated by $(\rho(u_{\mathbf{r}})+\sigma_1(u_{\mathbf{r}}, v_{\mathbf{r}})+\sigma_2(u_{\mathbf{r}}, v_{\mathbf{r}}))\cdot\bar{\sigma}_2(u_{\mathbf{a}}, v_{\mathbf{a}}/2, v_{\mathbf{a}}/2)$.

Relations (Total): A pair that passes all of the above forms a relation; the probability of this occurring is estimated by $(\rho(u_{\mathbf{r}}) + \sigma_1(u_{\mathbf{r}}, v_{\mathbf{r}}) + \sigma_2(u_{\mathbf{r}}, v_{\mathbf{r}})) \cdot (\rho(u_{\mathbf{a}}) + \sigma_1(u_{\mathbf{a}}, v_{\mathbf{a}}) + \sigma_2(u_{\mathbf{a}}, v_{\mathbf{a}}))$.

In §5.5 we evaluate the above for the NFS parameters sets of TWIRL.

5.2.5 Direct smoothness tests

To test the accuracy of the above ρ and σ_1-based estimates compared to the actual NFS yield, we tested $N_{\mathbf{r}}(a, b)$ and $N_{\mathbf{a}}(a, b)$ values for smoothness for wide ranges of (a, b) pairs.

In the absence of a sieve implementation capable of handling the range of factor base sizes we intended to test, we wrote a smoothness test that uses trial division up to 2^{30} combined with the Elliptic Curve Method factoring algorithm.

The simplest approach would have been to subject each successive number to be tested to trial division followed, if necessary, by the ECM. To obtain slightly greater speed, and without having to deal with the imperfections (overlooking some smooth values) and inconveniences (memory requirements, resieving or trial divisions to obtain the cofactor) of sieving, we used a different approach reminiscent TWIRL largish station design of §2.2.2.

The trial divisions were organized in such a way that a large consecutive range of a's could be handled reasonably efficiently, for a fixed b. For the algebraic norms this was achieved as follows (the rational norms are processed similarly). Let $[A_1, A_2]$ be a range of a-values to be processed. For every progression P_i corresponding to a pair (p_i, r_i) with $p < 2^{30}$, calculate the smallest $a_p \geq A_1$ such that $a_p \equiv br \bmod p$ (i.e., p divides $N_{\mathbf{a}}(a_p, b)$) and if $a_p \leq A_2$ insert the pair (p, a_p) in a heap that is ordered with respect to non-decreasing a_p values. Next, for $a = A_1, A_1 + 1, \ldots, A_2$ in succession compute $z = N_{\mathbf{a}}(a, b)$, remove all elements with $a_p = a$ from the top of the heap, divide out the corresponding factors p from z to obtain the cofactor $z' = \mu(2^{30}, z)$, and if $a_p + p \leq A_2$ insert $(p, a_p + p)$ in the heap. Each resulting z' has no factors smaller than 2^{30}; if $z' < 2^{60}$ then it is certainly prime, otherwise it is subjected to the ECM primality test. The full factorization is thus obtained and compared to the smoothness bounds. The analogous test is also done for $N_{\mathbf{r}}(a, b)$.[17]

[17] Due to the probabilistic nature of the ECM, factors between 2^{30} and the smoothness bound ($U_{\mathbf{a}}$ and $U_{\mathbf{r}}$) may be overlooked; but with proper ECM parameter settings and reasonably sized $U_{\mathbf{a}}$ and $U_{\mathbf{r}}$, such as the ones considered here, this occurs with negligible probability (and in any case, we will be erring on the side of caution by underestimating the yield).

5.3 Choice of NFS polynomials

5.3.1 Context

Computation of estimates, by either psixyological method or by direct smoothness testing, requires a concrete choice of the composite n, degree d, NFS polynomials $f(X)$ and $g(X)$, root m, skewness ratio ω and correction factor ξ (see §1.5.3). The choice of polynomials affects all parameters, since good polynomials (i.e., those that produces a high yield of relations) allow for smaller smoothness bounds and sieving regions. Finding such good polynomials is a non-trivial affair, and various methods have been developed to improve the yield by a noticeable factor compared to random polynomials fulfilling the necessary requirements [143][145][146][64].

Concentrating on 1024-bit composites (i.e., the most commonly used key size in the RSA cryptosystem) and 768-bit composites (as a more accessible milestone), the natural subjects for study are the RSA-1024 and RSA-768 challenge numbers published by RSA Data Security, Inc. (see §1.3.1). The advanced methods for polynomial selection have not been previously invoked for such large composites [105]. We have thus applied these techniques ourselves, with some necessitated minor adaptations, for the purpose of the ensuing evaluation. The results, given in these section and affecting the subsequent sections, have meanwhile been used by several follow-up works as well (see §6.3).

5.3.2 NFS polynomials for RSA-1024

The RSA-1024 factoring challenge from [177] is the following composite:

$n =$135066410865995223349603216278805969938881475605667027524485143851526510604859533833940287150571909441798207282164471551373680419703964191743046496589274256239341020864383202110372958725762358509643110564073501508187510676594629205563685529475213500852879416377328533906109750544334999811150056977236890927563

We have derived and evaluated numerous polynomials for this composite, using the NFS polynomial selection program of Jens Franke and Thorsten Kleinjung [64] (with minor adaptations). Our experiments employed several Pentium 1.7GHz computers at the Weizmann Institute of Science, for a total CPU time of about 20 days. Most of this time was spent on experimentation with search parameters, which can be reused for other composites, so future experiments would require just a few hours for polynomials of similar quality. Notably, with this polynomial selection program there is a lot of flexibility in the search parameters: at a small cost in yield one can obtain, for example, polynomials with much larger or much smaller skewness ratio ω, or trade root properties for size properties (see [146]).

The best polynomial pair found, and which was subsequently used for the parametrization of 1024-bit TWIRL in §2.2, is the following:

(A) $m =$ 262619868791250802778182368904158187239810594129624673885007610368230619674055292506154513387298663560887146183854975198160502278243245067074820593711054723850

$$570027395756140011420203134807117903732061718812827366825166704434650128222816083871694092824691383112595203927698431049857937444948214372729619704861,$$
$d = 5$, $s = 1991935.4$, $\xi = \exp(6.33)$,
$$\begin{aligned} f(X) = \; & 171930489423634514340101141808 0X^5 \\ & - 69919734888660586107407418604363447 1X^4 \\ & + 27086030483569532894050974257851346649521314X^3 \\ & + 46937584052668574502886791835536552277410242359042X^2 \\ & - 10107029484257211137178145885069684587770689954539450138 4X \\ & - 22666915939490940578617524677045371189128909899716560398434136, \\ g(X) = \; & 938772308370263069845713674770 27X \\ & - 37934895496425027513691045755639637174211483324451628365 \end{aligned}$$

Note that here we have a non-monic rational-side g, so compared to the special case given in §1.5.3 we redefine $N_{\mathbf{r}}(a,b) \stackrel{\text{def}}{=} |b \cdot g(a/b)|$.

For some of the other experiments, we used the following polynomials, obtained through the approach of Montgomery and Murphy [143][145][146]. This approach allows for $f(X)$ of degree other than $d = 5$, thereby allowing us to investigate the effect of varying the degree. However, it yields polynomials that are somewhat inferior to those of [64]. In these cases, $g(X) = X - m$. The best polynomials we found by this method, for d = 5,6,7,8,9, are as follows:

(B) $m = 401660614994057677612759225052058453196206732239623942698 48$, $d = 5$, $\omega = 87281.9$, $\xi = \exp(4.71)$,
$$\begin{aligned} f(X) = \; & 1291966090228800X^5 \; - \; 64092357265554977365242 1X^4 \\ & + 22084609569698872827347541432045436154518749958885X^3 \\ & + 39596889412070187463022609575354654771833433271171980 5X^2 \\ & - 96965973957066386285836042292532199420340774279358321957826X \\ & - 41492384851986578638826274128838175675496151871365204226808714 93 \end{aligned}$$

(C) $m = 62904286063558990272557233200273917159703450880 70$, $d = 6$, $\omega = 458.857$, $\xi = \exp(3.10)$,
$$\begin{aligned} f(X) = \; & 2180047385355840X^6 \; - \; 3142872579455569636X^5 \\ & - 12541556627968600362089925149698470015697 68X^4 \\ & - 12346184596682129311885354974311793670338999X^3 \\ & + 32685363049830158752687737781115278494499952052 2X^2 \\ & + 46093959111229794402396357057338090714782235467 68X \\ & - 11074692768758259967955017581674706364925519996590997 \end{aligned}$$

(D) $m = 10390029756781836031952464390691642545858 5$, $d = 7$, $\omega = 40.9082$, $\xi = \exp(3.66)$,
$$\begin{aligned} f(X) = \; & 10333080669249568440 00X^7 \; - \; 16075501154349035303847 9X^6 \\ & - 19530362723615105657667629630042775 1X^5 \; - \; 67322997660970472962322331424620518857X^4 \\ & + 85288668742268219444133849466758497928 3X^3 \\ & + 12226124738734620513750755416015521322344 9X^2 \\ & - 94104226259862845742589260929662484527821 8X \\ & - 38806712095590448575304126518627120637325432 \end{aligned}$$

(E) $m = 13648505386959137384028187687041215458$, $d = 8$, $\omega = 107.255$, $\xi = \exp(5.13)$,
$$\begin{aligned} f(X) = \; & 1121673850908090480 0X^8 && + 41269639628614893858 59X^7 \\ & - 11757919178224397829415075046 35X^6 && + 29966399990675338881961330352986 45X^5 \\ & + 20824014765601904804826252487710228 3X^4 && - 27357702926139861867857609251152887873X^3 \\ & - 34248340991002077428967269601147099265 35X^2 && - 12957538712647811491436510238283188219229X \\ & + 87332878299674868184413096619553988473477 05 \end{aligned}$$

(F) $m = 13107170715440628868594773605454 88$, $d = 9$, $\omega = 8.51584$, $\xi = \exp(3.89)$,
$$\begin{aligned} f(X) = \; & 11829510000X^9 && - 323042712742X^8 \\ & - 22960091664443611251501443 10X^7 && - 17667833832765445702215975840307X^6 \\ & + 10475098424346150979513979984790 8X^5 && + 68408289934182477896020018632506 4X^4 \\ & - 85584861328481518261781441424938636X^3 && + 32301718781994667946436083991144874X^2 \\ & - 42118837302218928303637260451515638X && - 12935588694082252819604375455691725 65 \end{aligned}$$

5.3.3 NFS polynomials for RSA-768

The RSA-768 factoring challenge from [177] is the following composite:

$n =$123018668453011775513049495838496272077285356959533479219732245215172640050726365751874520219978646938995647494277406384592519255732630345373154826850791702612214291346167042921431160222124047927473779408066535141959745985690214341 3

For RSA-768, we have derived the following polynomial pair (G) by the same method Franke-Kleinjung procedure as (A) above. Using the Montgomery-Murphy approach we have also obtained another (somewhat inferior) polynomial pair, denoted (H), with $d = 5$, $\omega \approx 26000$ and $\xi \approx \exp(5.3)$.[18]

(G) $m = 29804273545522569596216946680229697209251423355531365861703401903868659519214245843058509738994364817981329284550940228435757309840689061614767890670607800276082548461058468982659118338655899353388736496125545414357213967162299884 5$,
$d = 5$, $\omega = 1905116.1$, $\xi = \exp(3.78)$,
$f(X) = 4457235049589322 0X^5 + 1421806894351742986806319X^4 - 1319092270736482290377229028413X^3 - 4549121160536728229635596952173101064X^2 + 6062531470679201843447146909871507448641523X - 1814356642608474735992878928235210850251713945286$,
$g(X) = 669580586761796376057918067X - 7730028528962337116069068686542066657037329$

5.4 Results for extrapolated parameters

In this section, we estimate the main NFS parameters for 1024-bit and 768-bit composites using the traditional asymptotic extrapolation technique described in §5.2.2. We then apply the finer estimation techniques to these sizes, to investigate the soundness of these predictions and to obtain the remaining parameters.

5.4.1 Extrapolated parameters

5.4.1.1 Smoothness bounds and sieving region

Lenstra and Shamir [130] have proposed the following parameters for 512-bit numbers, in the context of TWINKLE:[19]

- 512-bit composites: $U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{24}$, $S = 1.6 \cdot 10^{16}$ ($A = 9 \cdot 10^9$, $B = 9 \cdot 10^5$)

[18] For incidental technical reasons, §5.4.2.2 uses (H) rather than (G).

[19] The concrete factorization project [44] which factored RSA-155 used essentially similar values in [44]. The details are complicated by the use of special-q lattice sieving and the variability of some parameters among contributors to the sieving effort.

Following §5.2.2, we first extrapolate to 1024-bit composites by increasing the smoothness bounds U, the number B of sieve lines and their length $R = 2A$, and the number of progressions (roughly $U/\ln(U)$) by a factor of $L_{2^{1024}}[1/3, (8/9)^{1/3}] \,/\, L_{2^{512}}[1/3, 2/3^{2/3}] \approx 2737$. We then adjust to a different choice of NFS parameters, corresponding to minimized throughput cost (instead of minimized run time; see Table 5.1).[20] This decreases the smoothness bounds by a factor of $\left(L_{2^{1024}}[1/3, (5/3)^{1/3}(2/3)] \,/\, L_{2^{1024}}[1/3, (8/9)^{1/3}]\right)^{-1} \approx 210$, but increases both the sieve line width R and the number of sieve lines B by a factor of $L_{2^{1024}}[1/3, (5/3)^{1/3}(5/6)]/L_{2^{1024}}[1/3, (8/9)^{1/3}] \approx 2.3$. The analogous computation is also carried out for 768-bit composites, yielding:

- 768-bit composites: $U_{\mathbf{r}} = U_{\mathbf{a}} = 1.2 \cdot 10^7$, $S = 4.2 \cdot 10^{20}$ ($A = 1.5 \cdot 10^{12}$, $B = 1.5 \cdot 10^8$)
- 1024-bit composites: $U_{\mathbf{r}} = U_{\mathbf{a}} = 2.5 \cdot 10^8$, $S = 6 \cdot 10^{23}$ ($A = 5.5 \cdot 10^{13}$, $B = 5.5 \cdot 10^9$)

If large primes and partial relations are not used (i.e., $\ell_{\mathbf{a}} = \ell_{\mathbf{r}} = 0$) then the uncertainty about cycle behavior disappears, but more sieving is needed. By simply taking the smoothness bound of 512-bit factorization to be 10^9 (which is the large primes bound used in [44]) and extrapolating as before, we then obtain this estimate:

- 1024-bit composites without partial relations: $U_{\mathbf{r}} = U_{\mathbf{a}} = 1.5 \cdot 10^{10}$, $S = 6 \cdot 10^{23}$

5.4.1.2 Matrix sizes

Focusing on 1024-bit composites, we extrapolate the matrix sizes for the cases of minimzed throughput cost and minimized running time.

Runtime-optimized matrix. In the factorization of RSA-155, the matrix had about 6.7 million columns [44].[21] Combining this figure with the $L(2/3^{2/3})$ matrix size growth rate corresponding to minimized time complexity, we obtain

$$6.7 \cdot 10^6 \;\cdot\; \frac{L_{2^{1024}}[1/3, 2/3^{2/3}]}{L_{2^{512}}[1/3, 2/3^{2/3}]} \;\approx\; 1.8 \cdot 10^{10} \;\;.$$

With some adjustment to account for the observed behavior of the $o(1)$ term, it is estimated that an optimal 1024-bit matrix would contain about $D = 10^{10}$ columns.

Column density is hard to predict due to the complicating preprocessing methods employed (see [43]). Since in RSA-155 the final matrix had average column density about 63, we assume here an average column density of $h = 100$.

[20] We choose the "Circuits" setting of Table 5.1, which is optimized under the assumption that the sieving step does not use much storage. This tends to increase the smoothness bounds compared to a more accurate optimization for throughput cost when sieving *is* employed; but as we shall see the extrapolated smoothness bounds are still too low.

[21] This matrix is, apparently, considerably smaller than optimal due to the choice of small smoothness bounds as necessitated by the then-available sieve implementations.

We refer to this matrix size as the *runtime-optimized matrix*. In Chapter 2 through Chapter 4 we also employ it as a generous overestimate to upper-bound the cost of processing a throughput-optimized matrix.[22]

Throughput-optimized matrix. By optimizing for throughput cost instead, i.e., correcting this matrix size for the $L((5/3)^{1/3}(2/3))$ matrix size growth rate for matrix exponent 5/2, we find

$$1.8 \cdot 10^{10} \cdot \frac{L_{2^{1024}}[1/3, (5/3)^{1/3}(2/3)]}{L_{2^{1024}}[1/3,2/3^{2/3}]} \approx 8.7 \cdot 10^7.$$

We arrive at an estimate of about $4 \cdot 10^7$ columns for the circuit-NFS 1024-bit matrix. We again, optimistically, assume that the average column density is about 100. We refer to this matrix as the *throughput-optimized matrix*.

5.4.2 Evaluation via smoothness probabilities

We proceed to evaluate the aforementioned parameters using the psixyological functions, as prescribed in §5.2.4. We shall see that while the base parameters for 512-bit composites are reasonable, their naive extrapolation (by the standard method) 768-bit and 1024-bit yields unrealistic values. Subsequently we shall explore NFS yields and costs in the parameter-space neighborhood of the extrapolated parameters, in order to find workable and effective adjustments, and to further the understanding of the tradeoffs.

5.4.2.1 512-bit composites

Let $n =$ RSA-155, $d = 5$, f and g as in RSA-155 [44], $\omega = 10800$, and $\xi = \exp(5.3)$. Application of our ρ and σ_1-based estimates to $U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{24}$, $V_{\mathbf{r}} = V_{\mathbf{a}} = 2^6 U_{\mathbf{r}} = 2^{30}$, $A = 9 \cdot 10^9$, and $B = 9 \cdot 10^5$ result in an estimated yield of $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})/8.9 \approx 2.4 \cdot 10^5$ *ff* relations[23], $2.2 \cdot 10^6$ *fp*'s, $9.1 \cdot 10^5$ *pf*'s, and $8.1 \cdot 10^6$ *pp*'s. Because the parameter choice was intended for the use of two large primes per side ($\ell_{\mathbf{a}}, \ell_{\mathbf{r}} = 2$), these results look acceptable: if more than one tenth of the matrix is filled with *ff*'s, combinations of multi-prime partial relations will certainly fill in the rest (as confirmed experimentally in the case of [44]).

In retrospect, the sieving effort (as defined in §5.2.1.2) could be made about 470 times lower, while keeping the same fraction of *ff* relations, by choosing larger smoothness bounds ($U_{\mathbf{r}} = 2^{29}$, $U_{\mathbf{a}} = 2^{30}$) and smaller region ($B = 4.0 \cdot 10^4$ and proportional A). However, sieving would have required a larger amount of accessible fast RAM than was available in 1999. Because $U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{24}$ is much smaller than the choice that would minimize the sieving effort, extrapolated parameters may result in a very large sieving effort as explained in §5.2.2.2. See also Table 5.4 below.

[22] This is needed because, as shall be soon shown, direct extrapolations for the throughput-optimized case do not yield a realistic result.

[23] *ff*, *fp*, etc. denote full-full, full-partial, etc. relations; see §1.5.3.

5.4.2.2 768-bit composites

We provide a very brief account of our evaluation of n = RSA-768 using the polynomial pair (H) mentioned in §5.3.3. The qualitative behavior is similar to the (more important) case of 1024-bit composites, discussed in much greater depth below.

To get $S = 2AB = 4.2 \cdot 10^{20}$ with $A = \omega B$, we set $B = 9 \cdot 10^7$. With $U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{24}$, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}}) = 2.1 \cdot 10^6$, and $V_{\mathbf{r}} = V_{\mathbf{a}} = 2^{10}U_{\mathbf{r}} = 2^{34}$ we estimate a yield of fewer than 40 *ff*'s, 1200 *fp*'s, 500 *pf*'s, and $2 \cdot 10^4$ *pp*'s. It is unlikely that this is sufficient, unless a substantial effort is spent on finding multi-prime partial relations. With increased smoothness bounds $U_{\mathbf{r}} = 2^{29}$, $U_{\mathbf{a}} = 2^{30}$, and the same sieving region, about $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})/16 \approx 5.2 \cdot 10^6$ *ff*'s can be expected, i.e., this setting is plausible given reasonable use of partial relations.

5.4.2.3 1024-bit composites

For n = RSA-1024 we considered the polynomial pairs (B)–(F), of degrees d = 5,6,7,8,9 respectively, each with corresponding integer m, skewness ratio ω, and correction factor ξ as specified in §5.3.2. For each of these degrees, the first block of Table 5.3 gives the estimated yield figures for the extrapolated region size $S = 6 \cdot 10^{23}$ and rounded[24] extrapolated smoothness bounds $U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{28}$. For the large prime bounds we list several possibilities, $V_{\mathbf{r}} = V_{\mathbf{a}} = 2^j U_{\mathbf{r}}$ for $j \in \{8,12,16\}$, and indicate the expected yield of *fp*, *pf*, and *pp* relations by fp_j, pf_j, and pp_j, respectively. Because the skewness ratio ω depends on d, the height $B = \sqrt{S/(2\omega)}$ and width $R = 2A = 2\omega B$ of the sieving region depend on d.

The second block of Table 5.3 gives the analogous results for increased smoothness bounds $U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{34}$, corresponding to extrapolation without partial relations (see §5.4.1.1).

Table 5.3 indicates that, unless multi-prime partial relations are collected on a much wider scale than customary or practical, the smoothness bounds obtained by extrapolation are infeasible. The increased smoothness bounds are insufficient as well, using the extrapolated sieving region size S, if partial relations are indeed not used.

Having observed that extrapolation did not yield acceptable results, we proceed to explore the neighborhood in parameter-space in order to better understand its behavior and obtain more realistic values.

Effect of sieving region. The two bottom blocks of Table 5.3 depict the effect of doubling and quadrupling A and B, thereby increasing $S = 2AB$ (and hence the sieving effort) by a factor 4 and 16, respectively. The number of *ff* relations still does not approach the goal $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})$.

To get the choice $U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{34}$ to work without partial relations, our estimates suggest that $d = 6$ with $B \approx 2.9 \cdot 10^{12}$ (corresponding to $S \approx 8 \cdot 10^{27}$) would suffice. This would, however, be about 13000 times more expensive than the asymptotic estimate: the initial $2.6 \cdot 10^{10}$ b-values

[24] We round up to a power of 2 for incidental technical reasons; the resulting overestimate of yield strengthens our conclusion.

Table 5.3: Estimated yields with polynomials (B)–(F) for extrapolated parameters

				$V_{\mathbf{r}} = V_{\mathbf{a}} = 2^8 U_{\mathbf{r}}$ ($j = 8$)			$V_{\mathbf{r}} = V_{\mathbf{a}} = 2^{12} U_{\mathbf{r}}$ ($j = 12$)			$V_{\mathbf{r}} = V_{\mathbf{a}} = 2^{16} U_{\mathbf{r}}$ ($j = 16$)		
d	ω	B	ff	fp_8	pf_8	pp_8	fp_{12}	pf_{12}	pp_{12}	fp_{16}	pf_{16}	pp_{16}
$U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{28}$, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}}) \approx 2.9\text{E}7$, $S = 6\text{E}23$, $W \approx 3.6\text{E}24$												
5	87281.9	1.9E9	22	4.7E2	2.3E2	5.1E3	9.2E2	4.3E2	1.8E4	1.6E3	6.9E2	5.1E4
6	458.9	2.6E10	74	1.7E3	6.3E2	1.4E4	3.3E3	1.1E3	5.0E4	5.8E3	1.8E3	1.4E5
7	40.9	8.6E10	1.5E2	3.6E3	1.0E3	2.4E4	6.9E3	1.8E3	8.1E4	1.2E4	2.8E3	2.2E5
8	107.3	5.3E10	34	8.2E2	1.8E2	4.5E3	1.6E3	3.2E2	1.5E4	2.8E3	4.8E2	4.0E4
9	8.5	1.9E11	3	69	14	2.5E2	1.3E2	24	8.2E2	1.8E2	37	2.2E3
$U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{34}$, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}}) \approx 1.5\text{E}9$, $S = 6\text{E}23$, $W \approx 3.8\text{E}24$												
5	87281.9	1.9E9	9.1E6	1.1E8	5.6E7	6.9E8	2.0E8	9.5E7	2.1E9	3.3E8	1.5E8	5.2E9
6	458.9	2.6E10	2.1E7	2.8E8	1.0E8	1.4E9	5.1E8	1.7E8	4.1E9	8.2E8	2.6E8	1.0E10
7	40.9	8.6E10	3.1E7	4.3E8	1.2E8	1.7E9	7.7E8	2.0E8	5.0E9	1.3E9	2.9E8	1.2E10
8	107.3	5.3E10	6.8E6	1.0E8	2.2E7	3.3E8	1.9E8	3.6E7	9.9E8	3.1E8	5.2E7	2.4E9
9	8.5	1.9E11	5.3E5	8.5E6	1.5E6	2.5E7	1.6E7	2.5E6	7.3E7	2.6E7	3.6E6	1.8E8
$U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{34}$, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}}) \approx 1.5\text{E}9$, $S = 2.4\text{E}24$, $W \approx 1.5\text{E}25$												
5	87281.9	3.7E9	1.9E7	2.4E8	1.2E8	1.5E9	4.4E8	2.0E8	4.6E9	7.1E8	3.1E8	1.1E10
6	458.9	5.1E10	4.0E7	5.6E8	2.0E8	2.7E9	1.0E9	3.3E8	8.1E9	1.6E9	5.0E8	2.0E10
7	40.9	1.7E11	5.2E7	7.4E8	2.0E8	2.9E9	1.3E9	3.3E8	8.7E9	2.2E9	5.0E8	2.1E10
$U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{34}$, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}}) \approx 1.5\text{E}9$, $S = 9.8\text{E}24$, $W \approx 6.1\text{E}25$												
5	87281.9	7.4E9	4.1E7	5.3E8	2.5E8	3.3E9	9.5E8	4.3E8	1.0E10	1.5E9	6.6E8	2.5E10
6	458.9	1.0E11	7.5E7	1.0E9	3.7E8	5.2E9	2.0E9	6.3E8	1.6E10	3.2E9	9.5E8	3.9E10
7	40.9	3.4E11	8.6E7	1.3E9	3.4E8	5.0E9	2.3E9	5.7E8	1.5E10	3.8E9	8.4E8	3.7E10

Table 5.4: Sieving effort to find $\Pi(2^i, 2^{i+1})/32$ *ff*'s for $d = 6$

i	effort	i	effort	i	effort	i	effort	i	effort	i	effort
28	1.5E36	32	5.6E26	36	1.7E23	40	4.8E21	44	1.2E21	48	9.6E20
29	4.7E32	33	3.7E25	37	5.2E22	41	2.9E21	45	1.0E21	49	1.0E21
30	1.4E30	34	4.2E24	38	2.0E22	42	2.0E21	46	9.4E20	50	1.2E21
31	1.7E28	35	7.2E23	39	9.1E21	43	1.5E21	47	9.3E20	51	1.4E21

produce about $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})/72$ *ff*'s, but the performance deteriorates for larger b's so that much more than 72 times the initial effort is needed to find $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})$ *ff*'s. For $d = 5$ or 7 it would be 1.1 or 3.5 times even more expensive, respectively.

Effect of partial relations. Using partial relations is probably a more efficient way to get $U_{\mathbf{r}} = U_{\mathbf{a}} = 2^{34}$ to work, as suggested by the last two parts of Table 5.3. There are no adequate methods yet to predict if the listed partial relation yield, perhaps augmented with partial relations with 3 or more large primes, would suffice or not; thus we cannot make any definite statements on the resulting cost or the semismoothness bound that would be required.

Effect of smoothness bounds. In Table 5.4 the effect of low smoothness bounds is illustrated. The total expected sieving effort to find $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})/32$ *ff*'s is listed for $d = 6$, $U_{\mathbf{r}} = 2^i$ with $i = 28,29,\ldots,51$ and $U_{\mathbf{a}} = 2U_{\mathbf{r}}$. The optimum $9.3 \cdot 10^{20}$ is achieved at $i = 47$. When i gets smaller the effort at first increases slowly and gradually, but around $i = 39$ the effort grows faster than the smoothness bounds shrink (so the throughput cost of sieving), and for smaller i the performance deteriorates rapidly — lending support to the caveat in §5.2.2.2.

Effect of smoothness bounds and relative cycle yield. Suppose that the (unknown) ratio

Table 5.5: Minimal sieving efforts to find $T(2^{i_\mathbf{r}},2^{i_\mathbf{a}})/c$ *ff*'s

	$c=1$		$c=8$		$c=16$		$c=32$		$c=64$		$c=128$	
d	$i_\mathbf{r}, i_\mathbf{a}$	effort	$i_\mathbf{r}, i_\mathbf{a}$	effort	$i_\mathbf{r}, i_\mathbf{a}$	effort	$i_\mathbf{r}, i_\mathbf{a}$	effort	$i_\mathbf{r}, i_\mathbf{a}$	effort	$i_\mathbf{r}, i_\mathbf{a}$	effort
6	48,49	1.6E23	47,48	7.2E21	47,48	2.6E21	47,48	9.2E20	47,48	3.3E20	46,47	1.2E20
7	47,49	9.4E22	47,49	3.5E21	46,48	1.1E21	46,47	3.5E20	45,47	1.1E20	45,46	3.6E19
8	48,50	3.7E23	47,49	1.0E22	46,48	3.0E21	46,48	8.7E20	45,47	2.5E20	45,47	7.5E19

Table 5.6: Estimated yields of polynomial (A) for extrapolated parameters

		$V_\mathbf{r}=V_\mathbf{a}=2^8U_\mathbf{r}$ $(j=8)$			$V_\mathbf{r}=V_\mathbf{a}=2^{12}U_\mathbf{r}$ $(j=12)$			$V_\mathbf{r}=V_\mathbf{a}=2^{16}U_\mathbf{r}$ $(j=16)$		
B	*ff*	fp_8	pf_8	pp_8	fp_{12}	pf_{12}	pp_{12}	fp_{16}	pf_{16}	pp_{16}
$U_\mathbf{r}=U_\mathbf{a}=2^{28}$, $\Pi(U_\mathbf{r},U_\mathbf{a})\approx 2.9\text{E}7$, $S=6\text{E}23$, $W\approx 3.6\text{E}24$										
3.88E8	9.9E2	2.0E4	9.7E3	2.0E5	3.8E4	1.8E4	6.8E5	6.6E4	2.8E4	1.9E6
$U_\mathbf{r}=U_\mathbf{a}=2^{34}$, $\Pi(U_\mathbf{r},U_\mathbf{a})\approx 1.5\text{E}9$, $S=6\text{E}23$, $W\approx 3.8\text{E}24$										
3.88E8	1.8E8	2.1E9	1.0E9	1.2E10	3.7E9	1.7E9	3.5E10	5.9E9	2.6E9	8.6E10
$U_\mathbf{r}=U_\mathbf{a}=2^{34}$, $\Pi(U_\mathbf{r},U_\mathbf{a})\approx 1.5\text{E}9$, $S=2.4\text{E}24$, $W\approx 1.5\text{E}25$										
3.88E8	3.8E8	4.5E9	2.2E9	2.5E10	8.1E9	3.7E9	7.7E10	1.3E10	5.6E9	1.9E11
$U_\mathbf{r}=U_\mathbf{a}=2^{34}$, $\Pi(U_\mathbf{r},U_\mathbf{a})\approx 1.5\text{E}9$, $S=9.8\text{E}24$, $W\approx 6.1\text{E}25$										
3.88E8	8.2E8	9.9E9	4.7E9	5.7E10	1.8E10	8.0E9	1.7E11	2.9E10	1.2E10	4.2E11

between the number of cycles and the number of *ff* relations is c. What smoothness bounds $U_\mathbf{r}=2^{i_\mathbf{r}}$ and $U_\mathbf{a}=2^{i_\mathbf{a}}$ will minimize the sieving effort needed to find the $\Pi(2^{i_\mathbf{r}},2^{i_\mathbf{a}})/c$ requisite *ff*'s? Table 5.5 shows the optimal choice, and corresponding sieving effort, for various c and degrees d. As expected, both effort and smoothness bounds decrease with increasing c. This effect is stronger for larger d. Overall, $d=7$ is the best choice, with $d=6$ better than $d=8$ for small c but vice versa for larger ones. For suboptimal smoothness bounds, however, $d=7$ may not be the best choice, as illustrated in Table 5.3.

Using the Franke-Kleinjung procedure. The above makes use only of polynomials (B)-(F), which were found via the Montgomery-Murphy procedure, in order to obtain a meaningful comparison between polynomials of different degrees. However, the degree-5 polynomial pair (A) which was found via the (adapted) Franke-Kleinjung procedure, gives better results due to its better root and size properties (see [146]). The corresponding yield estimates are given in Table 5.6, organized analogously to Table 5.3.

Note that Table 5.3 gives strong indication that degree $d=5$ is suboptimal, but the method we used to generate (A) is limited to $d=5$. One can expect that an adaptation of the improved algorithm to $d=6$ or $d=7$ will yield even better results. In this light, the parameter choices chosen for TWIRL in §5.5 according to polynomial (A) merely imply an upper bound on cost; further improvement is likely to be possible.

Effect of much better polynomials. In an actual factorization attempt considerably more time would be spent to find good polynomials, so we may expect polynomials better than (B)–(F), and even (A), to be found. We approximate this by applying our estimates to polynomials (B) and (C) but artificially adjusting the correction factors ξ to a larger value. In Table 5.7 the results are given if ξ is replaced by ξ^3 for $d=6$,7, with parameters as in Table 5.3. Using the

Table 5.7: Estimated yields for extrapolated parameters with correction factor ξ^3

				$V_\mathbf{r} = V_\mathbf{a} = 2^8 U_\mathbf{r}$ ($j = 8$)			$V_\mathbf{r} = V_\mathbf{a} = 2^{12} U_\mathbf{r}$ ($j = 12$)			$V_\mathbf{r} = V_\mathbf{a} = 2^{16} U_\mathbf{r}$ ($j = 16$)		
d	ω	B	ff	fp_8	pf_8	pp_8	fp_{12}	pf_{12}	pp_{12}	fp_{16}	pf_{16}	pp_{16}
$U_\mathbf{r} = U_\mathbf{a} = 2^{28}$, $\Pi(U_\mathbf{r}, U_\mathbf{a}) \approx 2.9\text{E}7$, $S = 6\text{E}23$, $W \approx 3.6\text{E}24$												
6	458.9	2.6E10	2.6E2	5.8E3	2.2E3	4.9E4	1.1E4	4.0E3	1.7E5	2.0E4	6.3E3	4.7E5
7	40.9	8.6E10	6.8E2	1.5E4	4.6E3	1.0E5	2.9E4	8.0E3	3.5E5	5.1E4	1.2E4	9.5E5
$U_\mathbf{r} = U_\mathbf{a} = 2^{34}$, $\Pi(U_\mathbf{r}, U_\mathbf{a}) \approx 1.5\text{E}9$, $S = 6\text{E}23$, $W \approx 3.8\text{E}24$												
6	458.9	2.6E10	5.7E7	7.2E8	2.8E8	3.5E9	1.3E9	4.9E8	1.1E10	2.1E9	7.3E8	2.6E10
7	40.9	8.6E10	9.9E7	1.3E9	3.9E8	5.1E9	2.4E9	6.4E8	1.5E10	3.9E9	9.4E8	3.7E10
$U_\mathbf{r} = U_\mathbf{a} = 2^{34}$, $\Pi(U_\mathbf{r}, U_\mathbf{a}) \approx 1.5\text{E}9$, $S = 2.4\text{E}24$, $W \approx 1.5\text{E}25$												
6	458.9	5.1E10	1.1E8	1.4E9	5.3E8	6.9E9	2.5E9	8.9E8	2.1E10	4.1E9	1.3E9	5.1E10
7	40.9	1.7E11	1.7E8	2.3E9	6.6E8	9.1E9	4.2E9	1.1E9	2.7E10	6.8E9	1.6E9	6.5E10
$U_\mathbf{r} = U_\mathbf{a} = 2^{34}$, $\Pi(U_\mathbf{r}, U_\mathbf{a}) \approx 1.5\text{E}9$, $S = 9.8\text{E}24$, $W \approx 6.1\text{E}25$												
6	458.9	1.0E11	2.0E8	2.8E9	1.0E9	1.4E10	4.9E9	1.7E9	4.1E10	8.0E9	2.6E9	1.0E11
7	40.9	3.4E11	2.8E8	4.0E9	1.1E9	1.6E10	7.3E9	1.9E9	4.8E10	1.2E10	2.8E9	1.2E11

Table 5.8: Estimated yields for asymptotically-extrapolated smoothness bounds with 6 polynomials

				$V_\mathbf{r} = V_\mathbf{a} = 2^8 U_\mathbf{r}$ ($j = 8$)				$V_\mathbf{r} = V_\mathbf{a} = 2^{12} U_\mathbf{r}$ ($j = 12$)			
d	ω	B	ff	fp_8	pf_8	pp_8	ECM effort	fp_{12}	pf_{12}	pp_{12}	ECM effort
$U_\mathbf{r} = U_\mathbf{a} = 2^{34}$, goal $\approx 5.3\text{E}9$, $S = 6\text{E}23$, $W \approx 1.9\text{E}24$											
6	458.9	2.6E10	1.3E8	1.7E9	6.2E8	8.2E9	$E5.6\text{E}20$	3.0E9	1.0E9	2.5E10	$E8.7\text{E}20$
7	40.9	8.6E10	1.8E8	2.6E9	7.1E8	9.9E9	$E3.6\text{E}21$	4.6E9	1.2E9	3.0E10	$E5.4\text{E}21$

current state-of-the-art of polynomial selection methods it is unlikely that such large correction factors can be found in practice. Thus, the figures in Table 5.7 are probably too optimistic. Compared to Table 5.3 the yield improves by a factor about 3: a relatively small effect that does not have an impact on the observations made above about $U_\mathbf{r} = U_\mathbf{a} = 2^{28}$ and $U_\mathbf{r} = U_\mathbf{a} = 2^{34}$. For example, for $d = 6$ and $U_\mathbf{r} = U_\mathbf{a} = 2^{34}$ not using partial relations (and correction factor ξ^3) would require $B = 9.4 \cdot 10^{11}$ with corresponding $S = 8.2 \cdot 10^{26}$; this is about 1300 times more expensive than the asymptotic extrapolations. Hence, our limited polynomial search does detract from the conclusion that direct extrapolation does not directly yield realistic parameter choices.

Effect of Coppersmith's NFS variant. We estimated the yield and performance of Coppersmith's multi-polynomial version of the NFS (see §5.2.1.3) by assuming that for any degree d we can find a set G of sufficient cardinality consisting of degree d polynomials with a shared root m modulo n and with skewness ratios and correction factors comparable to (B)-(F) in §5.3. Table 5.8 lists some estimates for $d = 6{,}7$ and $|G| = 6$, to Table 5.3. The dimension of the matrix increases 7/2-fold and the yield improves by a factor 6. The *fp* and *pp* yield increase may not be very effective, since large primes match only if they occur in the norm of the same polynomial f. The relation collection effort changes from sieving effort $W \approx 3.8 \cdot 10^{24}$ to sieving effort $W \approx 1.9 \cdot 10^{24}$ plus a number of semismoothness tests (indicated by "ECM effort"), involving the constant of proportionality Υ measuring the relative performance compared to sieving. As explained in §5.2.1.3, the implications depend on the efficiency of direct smoothness testing vs. sieving.

Table 5.9: Actual and estimated number of $(2^i,2^j,1)$-semismooth $N_{\mathbf{r}}(a,b)$'s for $d=6$

	i						
j	24	25	26	27	28	29	30
24	2.4E3(2.7E3)						
25	4.9E3(5.5E3)	6.3E3(7.0E3)					
26	7.7E3(8.6E3)	1.2E4(1.4E4)	1.5E4(1.7E4)				
27	1.1E4(1.2E4)	1.9E4(2.1E4)	2.8E4(3.1E4)	3.4E4(3.7E4)			
28	1.4E4(1.6E4)	2.6E4(2.9E4)	4.3E4(4.7E4)	6.1E4(6.7E4)	7.1E4(7.7E4)		
29	1.8E4(2.0E4)	3.4E4(3.7E4)	5.8E4(6.4E4)	8.9E4(9.8E4)	1.2E5(1.3E5)	1.4E5(1.5E5)	
30	2.2E4(2.4E4)	4.2E4(4.7E4)	7.5E5(8.2E4)	1.2E5(1.3E5)	1.8E5(1.9E5)	2.3E5(2.5E5)	2.5E5(2.7E5)
31	2.6E4(2.9E4)	5.1E4(5.7E4)	9.3E4(1.0E5)	1.5E5(1.7E5)	2.4E5(2.6E5)	3.3E5(3.6E5)	3.8E5(4.1E5)
32	3.1E4(3.4E4)	6.1E4(6.8E4)	1.1E5(1.2E5)	1.9E5(2.1E5)	3.0E5(3.2E5)	4.3E5(4.7E5)	5.1E5(5.5E5)
33	3.6E4(4.0E4)	7.2E4(8.0E4)	1.3E5(1.5E5)	2.3E5(2.5E5)	3.7E5(4.0E5)	5.5E5(5.9E5)	6.5E5(7.1E5)
34	4.2E4(4.7E4)	8.4E4(9.3E4)	1.6E5(1.7E5)	2.7E5(3.0E5)	4.4E5(4.8E5)	6.7E5(7.2E5)	8.0E5(8.7E5)
35	4.8E4(5.4E4)	9.7E4(1.1E5)	1.8E5(2.0E5)	3.2E5(3.5E5)	5.2E5(5.6E5)	8.0E5(8.6E5)	9.7E5(1.0E6)
36	5.5E4(6.1E4)	1.1E5(1.2E5)	2.1E5(2.3E5)	3.6E5(4.0E5)	6.0E5(6.5E5)	9.3E5(1.0E6)	1.1E6(1.2E6)
37	6.3E4(7.0E4)	1.3E5(1.4E5)	2.4E5(2.6E5)	4.2E5(4.6E5)	6.9E9(7.5E5)	1.1E6(1.2E6)	1.3E6(1.4E6)
38	7.1E4(7.9E4)	1.4E5(1.6E5)	2.7E5(3.0E5)	4.7E5(5.2E5)	7.8E5(8.5E5)	1.2E6(1.3E6)	1.5E6(1.6E6)
39	8.1E4(9.0E4)	1.6E5(1.8E5)	3.0E5(3.3E5)	5.3E5(5.8E5)	8.9E5(9.7E5)	1.4E6(1.5E6)	1.7E6(1.9E6)
40	9.1E4(1.0E5)	1.8E5(2.0E5)	3.4E5(3.8E5)	6.0E5(6.6E5)	1.0E6(1.1E6)	1.6E6(1.7E6)	1.9E6(2.1E6)

5.4.3 Evaluation via actual smoothness tests

The accuracy of our ρ and σ_1-based estimates as derived for $n=$ RSA-1024 was tested by applying smoothness tests (as explained in §5.2) to $N_{\mathbf{r}}(a,b)$ and $N_{\mathbf{a}}(a,b)$-values for wide ranges of (a,b)-pairs with coprime a and b and degrees and parameters as in §5.3. More than 100 billion values have been tested for degrees 6 and 7. No major surprises or unexpected anomalies were detected.

For $d=6$ this is illustrated in Tables 5.9, 5.10, and 5.11. Tables 5.9 and 5.10 contain the accumulated results of smoothness tests for $N_{\mathbf{r}}(a,b)$ and $N_{\mathbf{a}}(a,b)$-values, respectively, for more than 100 billion coprime (a,b) pairs and 176 different b values ranging from 2^9 to 2^{31}. They list the number of $(2^i,2^j,1)$-semismooth $N_{\mathbf{r}}(a,b)$ and $N_{\mathbf{a}}(a,b)$-values (for i,j ranges as specified in the tables) that were found using trial division up to 2^{30}. The $(\rho+\sigma_1)$-based are given in parentheses. Table 5.11 continues 5.10 for larger bounds; since such tests take longer, 5.6 million coprime and 13 different b-values ranging from 2^{14} to 2^{26}.

The estimated value is systematically slightly higher than the actual value; this can be attributed to the fact that the estimated values average over all positive numbers less than some bound, whereas most values that are actually tested are close to the bound. This is partly offset by the use of asymptotic smoothness probabilities, which are somewhat smaller than the concrete probabilities.[25]

For $d=7$ we found comparable results. Because of the asymptotic nature of the estimates, we expect their accuracy to improve for the larger sieving regions suggested by Table 5.3.

[25]For $\rho(u_{\mathbf{r}})$ the correction term is roughly $+0.423\rho(v_{\mathbf{r}}-1)/\log N_{\mathbf{r}}(a,b)$; see [14].

Table 5.10: Actual and estimated number of $(2^i,2^j,1)$-semismooth $N_{\mathbf{a}}(a,b)$'s for $d=6$

	i						
j	24	25	26	27	28	29	30
28	0(0.15)	0(0.41)	0(0.96)	0(1.85)	0(2.53)		
29	0(0.19)	1(0.54)	1(1.36)	1(2.94)	1(5.32)	1(7.01)	
30	0(0.23)	1(0.69)	1(1.80)	1(4.14)	1(8.34)	5(14.18)	10(16.87)
31	0(0.29)	1(0.86)	2(2.28)	2(5.45)	5(11.63)	17(21.94)	24(28.52)
32	1(0.34)	2(1.04)	3(2.81)	3(6.88)	8(15.21)	27(30.34)	40(41.10)
33	1(0.41)	2(1.24)	5(3.40)	5(8.45)	12(19.11)	39(39.44)	58(54.70)
34	1(0.48)	2(1.47)	5(4.05)	5(10.17)	15(23.36)	49(49.31)	70(69.41)
35	1(0.56)	2(1.72)	6(4.76)	7(12.05)	21(28.00)	60(60.01)	82(85.33)
36	1(0.65)	2(2.00)	7(5.55)	10(14.12)	27(33.05)	71(71.63)	97(102.57)
37	1(0.75)	2(2.30)	8(6.42)	11(16.39)	31(38.58)	82(84.26)	111(121.26)
38	2(0.86)	3(2.65)	9(7.38)	12(18.88)	36(44.61)	95(97.98)	132(141.52)
39	2(0.99)	3(3.03)	10(8.45)	14(21.62)	41(51.20)	106(112.90)	148(163.51)
40	2(1.13)	3(3.46)	11(9.62)	19(24.63)	47(58.41)	115(129.13)	163(187.36)

Table 5.11: Actual and estimated number of $(2^i,2^j,1)$-semismooth $N_{\mathbf{a}}(a,b)$'s for $d=6$ (cont.)

	i									
j	31	32	33	34	35	36	37	38	39	40
34	0(0.30)	0(0.49)	0(0.70)	0(0.82)						
35	0(0.39)	0(0.66)	0(1.03)	1(1.41)	1(1.62)					
36	0(0.48)	0(0.85)	0(1.38)	1(2.05)	1(2.73)	1(3.08)				
37	0(0.58)	0(1.05)	0(1.75)	2(2.72)	2(3.90)	2(5.03)	2(5.60)			
38	0(0.69)	0(1.26)	0(2.15)	2(3.44)	3(5.14)	4(7.11)	5(8.95)	5(9.84)		
39	1(0.81)	1(1.49)	1(2.58)	3(4.21)	4(6.46)	8(9.30)	9(12.48)	12(15.36)	13(16.72)	
40	1(0.93)	1(1.74)	1(3.04)	4(5.02)	6(7.86)	12(11.62)	15(16.20)	18(21.16)	21(25.51)	23(27.52)

5.5 The TWIRL sieving parameters

In order to increase relations yield and improve cost, the proposed TWIRL design (see §2.2) deviates from the aforementioned extrapolated parameters. We experimented with numerous parameter choices and polynomials, guided by the behavior observed in the preceding section, and evaluated via the psixyological smoothness functions as in §5.2.4 and a numerical model of TWIRL's cost. Crucially for the design of the TWIRL architecture and for upper-bounding the load in its auxiliary steps, we also obtain a wealth of additional information on intermediate candidates by employing the techniques of §5.2.4.3.

5.5.1 Yields for RSA-1024

Compared to the previous section, TWIRL uses higher smoothness bounds: $U_{\mathbf{r}} = 3.5 \cdot 10^9$, $U_{\mathbf{a}} = 2.6 \cdot 10^{10}$, $V_{\mathbf{r}} = 4.0 \cdot 10^{11}$, $V_{\mathbf{a}} = 6.0 \cdot 10^{11}$.[26] Also, the number of large primes is set to $\ell_{\mathbf{r}} = \ell_{\mathbf{a}} = 2$. Conversely, the sieving region size is reduced to $S = 3.0 \cdot 10^{23}$. Table 5.12 gives the corresponding estimates of yield for the various kinds of full and partial relations. It also lists the predicted number of intermediate candidates, using the notation of §5.2.4.3.

[26] This has a dramatic effect, suggesting that the result of the asymptotic extrapolation indeed resides on the steep region of the run-time curve (see §5.2.2.2).

Table 5.12: RSA-1024 parameter sets for TWIRL with 130nm process technology
$U_{\mathbf{r}} = 3.5\text{E}9$, $U_{\mathbf{a}} = 2.6\text{E}10$, $V_{\mathbf{r}} = 4.0\text{E}11$, $V_{\mathbf{a}} = 6.0\text{E}11$, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}}) \approx 1.3\text{E}9$, $S = 3.0\text{E}23$, $d = 5$, $\omega = 1991935.4$, $B = 2.7\text{E}8$

$(\ell'_{\mathbf{a}}, \ell'_{\mathbf{r}})$-partial relations yield	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)	Total
	5.6E7	3.0E8	6.7E8	3.1E8	1.7E9	3.8E9	6.6E8	3.5E9	7.9E9	1.9E10

#PRS	#PBS	#PPT	#RCF	#RSS	#ACF	avg($N_{\mathbf{r}}(a,b)$)	avg($N_{\mathbf{a}}(a,b)$)
1.1E20	5.0E12	6.2E10	4.9E10	3.4E10	2.7E10	5.2E63	3.1E103

Ultimately we are interested in the number of cycles among the relations found. Alas, the dependence of the number of cycles on the number (and type) of relations is poorly understood (see §1.5.3, §5.2.1.1). As noted, $\pi(V_{\mathbf{r}}) + \pi(V_{\mathbf{a}})$ relations always suffice, and in past experiments the number of relations collected was always somewhat lower. Here, the estimated number of relations is a reasonable $0.49 \cdot (\pi(V_{\mathbf{r}}) + \pi(V_{\mathbf{a}}))$. Using $\ell_{\mathbf{a}}, \ell_{\mathbf{r}} > 2$ would further increase the relation yield. Note that there are $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})/23.2$ *ff*'s, which seems fairly generous compared to past experiments.

Notably, while the most "fertile" area of the sieving region is close to the origin (small a and b), unlike the situation in §5.4.2.3, here the relation yield has not yet "dried out": for example, doubling the region size S to $6 \cdot 10^{23}$ increases the number of relations significantly, to $2.8 \cdot 10^{10}$. The practical significance is that if someone builds a TWIRL device with hard-wired smoothness bounds and (for any reason) does not find enough relations using the above parameters, recovery may be possible simply by increasing S, i.e., by sieving for a longer time using the same hardware.

5.5.2 Candidates yield in TWIRL

The candidates yield data in Table 5.12 lets us analyze various stages in the sieving and relation collection can be likewise analyzed. For example the #PRS value is used for the cascaded sieves of TWIRL §2.3.5.

Of particular concern is *cofactor factorization*, i.e., the factoring of (not too) large norms after dividing away small factors that were discovered during sieving. We expect to perform about such $\#\text{RCF} + \#\text{ACF} = 7.7 \cdot 10^{10}$ factorizations, for integers whose size is at most $\max(V_{\mathbf{r}}, V_{\mathbf{a}})^2 = 3.6 \cdot 10^{23}$. Such factorizations require under 30ms on average using a modern CPU. Thus, the cofactor factorization can be completed in 1 year (i.e., in parallel to the operation of the TWIRL device) using under 74 bare-bones PCs. This cost is negligible compared to the cost of TWIRL, and in large volumes custom hardware would reduce it further.

5.5.3 Optimality and effect of technological progress

The TWIRL parameters were determined by practical concerns. Most crucially, they employ the largest value of $U_{\mathbf{a}}$ for which the algebraic-side TWIRL device still fits on single silicon wafer. Theoretically, this $U_{\mathbf{a}}$ is suboptimal; it would be beneficial to increase it further. Such increase will become possible when progress in chip manufacturing technology allows fitting larger circuits into a single wafer, either by increasing the wafer size or by decreasing the feature size. Thus, for the near future we may expect the cost of TWIRL to decrease more than linearly as a function of the relevant technological parameters, i.e., faster than naively implied by Moore's law.

Table 5.13: RSA-1024 parameter sets for TWIRL with 90nm process technology

$U_{\mathbf{r}} = 1.2\text{E}10$, $U_{\mathbf{a}} = 5.5\text{E}10$, $V_{\mathbf{r}} = 8.0\text{E}11$, $V_{\mathbf{a}} = 1.0\text{E}12$, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}}) \approx 2.9\text{E}9$, $S = 8.0\text{E}22$ $d = 5$, $\omega = 1991935.4$, $B = 1.4\text{E}8$

$(\ell'_{\mathbf{a}}, \ell'_{\mathbf{r}})$-partial	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)	Total
relations yield	2.2E8	9.8E8	1.8E9	9.2E8	4.0E9	7.5E9	1.4E9	6.1E9	1.1E10	3.4E10

#PRS	#PBS	#PPT	#RCF	#RSS	#ACF	avg($N_{\mathbf{r}}(a,b)$)	avg($N_{\mathbf{a}}(a,b)$)
6.3E19	1.1E13	9.8E10	7.2E10	5.9E10	4.5E10	2.7E63	1.1E102

$U_{\mathbf{r}} = 1.2\text{E}10$, $U_{\mathbf{a}} = 5.5\text{E}10$, $V_{\mathbf{r}} = 9.0\text{E}11$, $V_{\mathbf{a}} = 1.2\text{E}12$, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}}) \approx 2.9\text{E}9$, $S = 7.3\text{E}23$ $d = 5$, $\omega = 1991935.4$, $B = 4.3\text{E}8$

$(\ell'_{\mathbf{a}}, \ell'_{\mathbf{r}})$-partial	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)	Total
relations yield	7.9E8	3.9E9	7.9E9	3.4E9	1.7E10	3.4E10	5.4E9	2.7E10	5.5E10	1.5E11

#PRS	#PBS	#PPT	#RCF	#RSS	#ACF	avg($N_{\mathbf{r}}(a,b)$)	avg($N_{\mathbf{a}}(a,b)$)
5.2E20	4.6E13	4.6E11	3.4E11	2.7E11	2.1E11	8.1E63	2.8E104

Table 5.14: RSA-768 parameter sets for TWIRL

$U_{\mathbf{r}} = 1.0\text{E}8$, $U_{\mathbf{a}} = 1.0\text{E}9$, $V_{\mathbf{r}} = 2.0\text{E}10$, $V_{\mathbf{a}} = 3.0\text{E}10$, $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}}) \approx 5.7\text{E}7$, $S = 3.0\text{E}20$ $d = 5$, $\omega = 1905116.1$, $B = 8.9\text{E}6$

$(\ell'_{\mathbf{a}}, \ell'_{\mathbf{r}})$-partial	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)	Total
relations yield	3.5E6	2.2E7	5.5E7	2.5E7	1.5E8	3.9E8	6.2E7	3.8E8	9.7E8	2.1E9

#PRS	#PBS	#PPT	#RCF	#RSS	#ACF	avg($N_{\mathbf{r}}(a,b)$)	avg($N_{\mathbf{a}}(a,b)$)
5.3E17	3.4E11	7.5E9	6.3E9	3.9E9	3.2E9	3.4E49	7.1E82

For a concrete example, one may consider an implementation of TWIRL using 90nm process technology (Custom-90-D from §3.7.1). According to ITRS [92], compared to the 130nm process technology considered in Chapter 2, we may assume a reduction in area by a factor of 2 and an increase in speed by a factor of 2, for a total cost reduction by a factor of 4. Table 5.13 presents two appropriate NFS parameter sets that make use of the denser wafers.

The first parmeter set is about as plausible as the one in Table 5.12, in terms of the assumptions on cycle behavior. The cost of such a TWIRL implementation is roughly 1.1M US$ $\times$ years (predicted analogously to [187]) — considerably lower than 2.5M US$ $\times$ years one may expect.

The second parameter set in Table 5.13 shows the effect of improved technology on yield, when keeping the cost constant at 10M US$×years (i.e., the same as in Chapter 2). Here, the estimated number of relations is $1.95 \cdot (\pi(V_{\mathbf{r}}) + \pi(V_{\mathbf{a}}))$, which is nearly twice the trivially sufficient number. Also, there are $\Pi(U_{\mathbf{r}}, U_{\mathbf{a}})/3.6$ *ff*'s, which is much more than in any recent factoring experiment. Thus, we may conclude that using 90nm technology, a budget of 10M US$×years per factorization (in large quantities) leaves an ample safety margin — arguably, more than enough to account for estimation errors, relations that are lost due to approximations in the sieving process, and sub-optimal cycles-finding algorithms.

5.5.4 Yields for RSA-768

We applied an analogous derivation, using smoothness probability estimates and a numerical TWIRL cost mode, to choose the parameters for 768-bit TWIRL. Table 5.14 lists these, along with the yield and candidates estimates using the RSA-768 polynomial pair (G).

Chapter 6

Conclusions and implications of Part I

6.1 Summary of results

In Chapter 2, we have presented TWIRL, a new special-purpose electronic sieving device for the Number Field Sieve. The device consists of a massively parallel pipeline that carries sieve locations through various stations, where they experience the addition of progression contributions in myriad different ways that are optimized for the varying scales of progression periods. In factoring 512-bit integers, the new device is 1,600 times faster than the best previously published designs. For 1024-bit composites, with appropriate choice of NFS parameters and modern 90nm VLSI technology, the new device can complete the sieving task within 1 year at an estimated cost of US$ 1.1, i.e., about a *millionth* of previous predictions.

In Chapter 3, we have presented a special-purpose device for the matrix-by-vector multiplications that occur in the block Wiedemann algorithm of the NFS linear algebra step. This mesh-based device dramatically improves the efficiency of the NFS linear algebra step compared to previous proposals. For 1024-bit composites, it brings the cost of NFS linear algebra step to well within practical range — albeit with some reservations regarding technological feasibility.

In Chapter 4, we resolved the aforementioned reservations by introducing a completely different architecture, based on a pipelined systolic approach, which exhibits several advantages over the prior (mesh-based) approach. It has a lower cost and modest technological requirements; specifically, unlike previous proposals it uses standard chip sizes and purely local communication. The architecture is scalable, and offers the flexibility to handle problems of varying sizes. Its operation is deterministic and allows local simulation and verification of components. We have also described an efficient error detection and recovery mechanism, which can also be adapted to other software or hardware implementations of Wiedemann's algorithm. Even with a conservative (i.e., overestimated) matrix size, this device can complete the sieving task within 1 year at an estimated cost of US$ 0.4.

In Chapter 5, we supported the previous chapters by analysis of pertinent NFS parameters. We have applied numerical methods to estimate the yield of the NFS when applied to 1024-bit and 768-bit RSA moduli, and tested the accuracy of our results using actual smoothness tests. We also generalized and extended the parameter estimation technique to support the finer points of our analysis.

Implications for RSA keys. It has been often claimed and relied upon, in the academia in industry, that 1024-bit RSA keys would resist cryptanalysis for at least an additional decade — arguing that both NFS sieving and the NFS matrix step would be unfeasible. Our results cast grave doubt on these claims: if the prescribed devices are built, they would break RSA keys in under one year at a cost of a few million US$. For 768-bit RSA keys, the amortized cost per key is just a few thousand US$.

Our results do not imply that breaking 1024-bit RSA is within reach of individual hackers. However, it is difficult to identify any specific issue that may prevent a sufficiently motivated and well-funded organization from applying the Number Field Sieve to 1024-bit composites within the next few years. It should be stressed that these conclusions are based on many assumptions necessitated by the incomplete understanding of the Number Field Sieve, and are not yet supported by an actual implementation. However, they form a strong indication as to the prudence of depending on the hardness of this step. Any security assessment of such keys should take this into account, and indeed our results are already reflected in, and referenced by, recent industry practice and standards (e.g., by the IEEE, Internet RFC editor, and several governments; see below).

6.2 Notes

Benefits of special-purpose hardware. An obvious benefit of special-purpose hardware is that the cost of any algorithm can be reduced (compared to a software implementation) by eliminating intermediate abstraction layers and discarding irrelevant peripheral hardware. One cryptanalytic example of this point is the EFF DES Cracker [59], which employed 36,864 dedicated chips to perform exhaustive search on a DES key, and did so at a small fraction of the cost (per unit of throughput) compared to similar experiments that used general-purpose computers.

However, special-purpose hardware can go well beyond efficient implementation of standard algorithms. Custom circuit design allows for specialized data paths, flexible partitioning of resources, enormous parallelism, and even for the use of non-electronic computing devices. Taking advantage of these requires new algorithms and adaptations, as has been evident in the preceding chapters.

Using semi-custom hardware. In an attempt to decrease the initial Non Recurring Engineering cost, one may consider semi-custom technologies such as Field Programmable Gate Array chips. This was shown to be unfavorable for one case in Chapter 3, where the FPGA-based implementation was significantly less efficient than both the custom designs and properly parallelized PC-based implementation. We have carried out similar evaluations for the other architectures

considered here, with similar results. The reason is twofold. The obvious one is the lower efficiency, in terms of computational power per circuit area, of FPGA vs. custom circuits (due to the higher abstraction level). The subtler but more essential reason is that all the architectures we propose in this work are severely bandwidth-limited[1], and FPGA chips do not provide sufficient resources per chip nor sufficient inter-chip bandwidth to efficiently support the relevant problem scales.

Balance between sieving and linear algebra. The cost estimates herein give strong indication that, for the parameter choices considered, the linear algebra step is easier than the sieving step. In light of this, one may try to improve the overall performance of NFS by re-balancing the relation collection step and the matrix step, i.e., by increasing the smoothness bounds to make sieving easier. However, the parameter choices employed for TWIRL in Chapter 2 and Chapter 5 are already close to optimal in terms of sieving cost. Also, as we explain in [131], one cannot expect such re-balancing to bring significant asymptotic improvement either.

6.3 Impact and follow-up works

The following is a brief survey of some of the works following up on the research presented in this part of the dissertation, and its impact on industry standards and practice.

Industry and government standards. Following its immediate consideration by industry (e.g., [97]), our research is now reflected and explicitly referenced by recent standards such as IEEE 1363a-2004 [88], RFC 3766 [159] and RFC 4359 [215], and the German federal criteria for electronic signatures (e.g., [175]). The relevant key size recommendations of NIST Special Publication 800-57 [152] were likewise revised compared to drafts [151] preceding our publications.

SHARCS workshop. This research served as a primary motivation and focus for a new annual workshop, SHARCS (Workshop on Special Purpose Hardware for Attacking Cryptographic Systems [220, 221, 222]), on whose program committees I serve since 2006.

Pertinent publications following up on ours include:

Distributed linear algebra. In [75], Geiselmann and Steinwandt proposed a distributed variant of our mesh-routing-based linear algebra device; for a discussion see §3.6.

FPGA implementation of routing-based device. In [18][19], Bajracharya et al. report on a detailed VHDL design for (one case of) the mesh-routing-based linear algebra device of Chapter 3, and the resulting costs, using Field Programmable Gate Arrays.

Improved mesh-based sieving. In [76], Geiselmann and Steinwandt proposed a mesh-based sieving architecture which addresses the cost and scalability issues of their prior design [74] (see §1.6.6). It incorporates several ideas from our works on routing-based linear algebra of Chapter 3

[1] For certain cryptanalytic algorithms this is indeed an provable bottleneck [217].

and the TWIRL architecture of Chapter 2; in particular, it employs clockwise transposition routing and compact DRAM-based progression representations. It also relies on the NFS parameter analysis of Chapter 5. The estimated cost of factoring 768-bit composites via this architecture, in terms of circuit area, is 6.3 times larger than of the TWIRL.[2]

Eliminating the diaries from TWIRL. In [69], Geiselmann and Steinwandt show that the diary components of TWIRL, described in §2.3.7.2, can be eliminated.[3] They showed that the information previously encoded in the diaries can be reconstructed at an acceptable cost using post-processing via a special-purpose implementations of the Elliptic Curve Method, described in that paper.

Another sieving architecture. In [77], Geiselmann and Steinwandt describe a special-purpose sieving device which resolves a drawback of the TWIRL device, namely the need for wafer-scale integration (at a cost of 2-3.5 increase in silicon area). They employ a pipelined approach which combines many of the ideas from Chapter 2 through Chapter 4, and the analysis in Chapter 5.

SHARK. In [65][66], Franke et al. present a highly-parallel electronic sieving device. Posed as an alternative to TWIRL, it uses smaller chips connected via a massive butterfly routing network. Unlike TWIRL, it is tailored for special-q lattice sieving. The cost estimate for 1024-bit composites is higher than TWIRL's ($200M vs. $1.1M for 1024-bit composites in one year), but the technological challenges differ.

[2] If one requires that the individual chip sizes are also the same and re-parametrized TWIRL accordingly, the gap is roughly halved.

[3] This component is responsible for just 3% of the ciruit size, but affects the circuit layout and the design's complexity.

Part II

Side-channel attacks

כְּבֹד אֱלֹהִים הַסְתֵּר דָּבָר; וּכְבֹד מְלָכִים חֵקֶר דָּבָר

— משלי כ"ה 2

It is God's privilege to conceal things
and the king's privilege to discover them.

— Proverbs XXV 2
(New Living Translation)

Chapter 7

Introduction

7.1 Overview of Part II

In this Part we describe new types of side-channel cryptanalytic attacks, and their application to major cryptosystems. Unlike many other side channels, these attacks are readily deployed and widely applicable. We analyze their potential, experimentally demonstrate their power and efficiency, and discuss countermeasures.

The remainder of Chapter 7 briefly surveys known side-channel attacks.

In Chapter 8 we describe pure software attacks that exploit inter-process information leakage through the state of the CPU's memory cache. We show how these attacks can be applied to the AES cipher, and demonstrate an efficient attack on OpenSSL and Linux's `dm-crypt` encrypted partitions. Some variants of our attack do not even require known plaintext or ciphertext, and have no direct interaction with the analyzed process other than running on the same CPU.

In Chapter 9 we describe attacks that exploits acoustic emanations from modern computers, recorded via a plain microphone. These emanations are correlated with processor activity and carry an unexpected wealth of information. We demonstrate acoustic leakage of secret information from sensitive computation such as RSA signing and decryption.

Chapter 10 summarizes the results of Part II and briefly surveys recent works which appeared after our published results.

7.2 Side-channel attacks

In the design of cryptographic algorithms and protocols, as in other computational settings, it is postulated that parties communicate via designated input and output channels. However, in practical adversarial settings we are often reminded that the designated channels are merely

convenient fiction: the underlying physical reality is that every concrete realization of a computational process interacts with its environment in unintended — and often unexpected — ways. From the environment's perspective, this creates additional, inadvertent inputs and outputs to the computation. In cryptanalysis, *side-channel attacks* are methods that exploit these unintentional information channels.

A large number of information-bearing side channels have been discovered and exploited cryptanalytically. Some of these, in particular those exploiting inadvertent inputs, require invasive access to the device (e.g., physical[1] fault induction [32, 27] or attachment of electric probes). Other channels are non-intrusive and rely solely on passive measurement; these include:[2]

- Radio-frequency electromagnetic emission ("TEMPEST") [111, 136].
- Modulation of electromagnetic radiation sources ('NONSTOP" [100]) such as a beam sent by the attacker [223], a cellular phone [11], or an RFID tag reader [158].
- Electrical signals leaking to unintended wires, such as power lines or unsecured communication lines [223].
- Diffuse visible light, such as light emitted by display devices or status LEDs and reflected by walls [111].
- Power-supply current fluctuations [109, 138, 137].
- Acoustic emanations from mechanical devices, such as mechanical cipher machines [223], dot-matrix printers [38], or computer keyboards and ATM keypads [13].
- Timing of events on otherwise-legitimate output channels (*timing attacks*) [108, 181, 100, 31] (see §7.3).

Despite this extensive catalog of attack vectors, it appears that side-channel attacks are limited in their practical security implications to modern computing equipment, since most of these side channels require special measurement equipment and an expert human operator, or are applicable only in very restricted circumstances (e.g., a smartcard in the hands of the attacker). Moreover, these attacks are by now well-recognized and often protected against.

7.3 Timing attacks

Herein we briefly describe the notion of *timing attacks*, a known technique which we employ in one of our acoustic attacks (see §9.2.4). This class of side-channel attacks, introduced by

[1] Another case of fault induction exploits algorithmic faults such as the possibility of decryption failure (shown by Proos for NTRU [171]; Dwork et al. [58] provide a generic algorithmic countermeasure). Strictly, this is not a side-channel attack since it does not violate the processing abstraction,

[2] Here we reference but a few of the seminal works; see [179] for an extensive bibliographic resource.

Kocher [108], exploits the fact that many computational operations vary in time depending on the inputs to the operation. By measuring the running time of the operation we learn something about its inputs. For example, in the RSA cryptosystem [176], decryption of a ciphertext c is done by treating c as a large number, and computing the power $c^d \pmod{n}$ where d is part of the secret key and n is the public key. The simplest (though inefficient) algorithm for computing this exponentiation is to multiply c by itself d times; this takes time proportional to d, so in this case measuring the decryption time will give an estimate of d. The algorithms used in practice (e.g., "square and multiply" with sliding windows and Montgomery multiplications) are much more efficient, but exhibit similar properties unless carefully designed to thwart such attacks. By combining many measurements that correspond to different properties of the key, the possibilities can be narrowed down until the key is fully recovered.

Boneh and Brumley [31] showed that, using timing information accurate to within approx. 1 millisecond, the secret key can be extracted from common RSA implementations. In the pertinent attack variants, the attacker submits chosen inputs (i.e., chosen ciphertexts when attacking RSA decryption and chosen messages when attacking RSA signing), and observes the time it takes the legitimate user to process (i.e., decrypt or sign) these inputs using the secret key. The latter requires a feedback channel. In [31], the queries were via a network to an OpenSSL-based server, and the attack code measured the time between the query and response packets.

Chapter 8

Efficient cache attacks on AES

שנים אוחזין בטלית
זה אומר אני מצאתיה וזה אומר אני מצאתיה
זה אומר כולה שלי וזה אומר כולה שלי
— בבא מציעא א,א

8.1 Introduction

8.1.1 Overview

Many computer systems concurrently execute programs with different privileges, employing various partitioning methods to facilitate the desired access control semantics. These methods include kernel vs. userspace separation, process memory protection, filesystem permissions and `chroot`, and various approaches to virtual machines and sandboxes. All of these rely on a model of the underlying machine to obtain the desired access control semantics. However, this model is often idealized and does not reflect many intricacies of the actual implementation.

In this chapter we show how a low-level implementation detail of modern CPUs, namely the structure of memory caches, causes subtle indirect interaction between processes running on the same processor. This leads to cross-process information leakage. In essence, the cache forms a shared resource which all processes compete for, and it thus affects and is affected by every process. While the *data* stored in the cache is protected by virtual memory mechanisms, the *metadata* about the contents of the cache, and in particular the memory access patterns of processes using that cache, is not fully protected.

We describe several methods an attacker can use to learn about the memory access patterns of another process, e.g., one which performs encryption with an unknown key. These are classified into methods that affect the state of the cache and then measure the effect on the running time of the encryption, and methods that investigate the state of the cache after or during encryption. The latter are found to be particularly effective and noise-resistant.

We demonstrate the cryptanalytic applicability of these methods to the Advanced Encryption Standard (AES, [149]) by showing a known-plaintext (or known-ciphertext) attack that performs efficient full key extraction. For example, an implementation of one variant of the attack performs full AES key extraction from the `dm-crypt` system of Linux using only 800 accesses to an encrypted file, 65ms of measurements and 3 seconds of analysis; attacking simpler systems, such as "black-box" OpenSSL library calls, is even faster at 13ms and 300 encryptions.

One variant of our attack has the unusual property of performing key extraction *without knowledge of either the plaintext or the ciphertext*. This is a particularly strong form of attack, which is clearly impossible in a classical cryptanalytic setting. It enables an unprivileged process, merely by accessing its own memory space, to obtain bits from a secret AES key used by another process, without any (explicit) communication between the two.This too is demonstrated experimentally.

Implementing AES in a way that is impervious to this attack, let alone developing an efficient generic countermeasure, appears non-trivial; in Section 8.5, various countermeasures are described and analyzed.

8.1.2 Related work

The possibility of cross-process leakage via cache state was first considered in 1992 by Hu [87] in the context of intentional transmission via covert channels. In 1998, Kelsey et al. [101] mentioned the prospect of "attacks based on cache hit ratio in large S-box ciphers". In 2002, Page [161] described theoretical attacks on DES via cache misses, assuming an initially empty cache and the ability to identify cache effects with very high temporal resolution in side-channel traces. He subsequently proposed several countermeasures for smartcards [162], though most of these require hardware modifications and are inapplicable or insufficient in our attack scenario. Recently, variants of this attack (termed "trace-driven" in [162]) were realized by Bertoni et al. [25] and Acıiçmez and Koç [3][4], using a power side channel of a MIPS microprocessor in an idealized simulation. By contrast, our attacks operate purely in software, and are hence of wider applicability and implications; they have also been experimentally demonstrated in real-life scenarios.

In 2002 and subsequently, Tsunoo et al. devised a timing-based attack on MISTY1 [206, 207] and DES [205], exploiting the effects of collisions between the various memory lookups invoked internally by cipher (as opposed to the cipher vs. attacker collisions we investigate, which greatly improve the attack's efficiency). Recently Lauradoux [115] and Canteaut et al. [41] proposed some countermeasures against these attacks, none of which are satisfactory against our attacks (see Section 8.5).

Concurrently with but independently of this work, Bernstein [23] described attacks on AES that exploit timing variability due to cache effects. This attack can be seen as a variant of our Evict+Time measurement method (see Section 8.3.4 and the analysis of Neve et al. [154]), though it is also somewhat sensitive to the aforementioned collision effects. The main difference is that [23] does not use an explicit model of the cache and active manipulation, but rather relies only on the existence of some consistent statistical timing pattern due to various memory access effects

which are neither controlled nor modeled. The resulting attack is simpler and more portable than ours, but has several shortcomings. First, it requires reference measurements of encryption under *known* key in an identical configuration, and these are often not readily available (e.g., a user may be able to write data to an encrypted filesystem, but creating a reference filesystem with a known key is a privileged operation). Second, the attack of [23] relies on timing the encryption and thus, similarly to our Evict+Time method, seems impractical on many real systems due to excessively low signal-to-noise ratio; our alternative methods (Sections 8.3.5 and 8.4) address this. Third, even when the attack of [23] works, it requires a much higher number of analyzed encryptions than our method.[1] The recent paper of Canteaut et al. [41] describes a variant of Bernstein's attack which focuses on internal collisions (following Tsunoo et al.) and provided a more in-depth experimental analysis; its properties and applicability are similar to Bernstein's attack.[2]

Also concurrently with but independently of our work, Percival [167] described a cache-based attack on RSA for processors with simultaneous multithreading. The measurement method is similar to one variant of our asynchronous attack (Section 8.4), but the cryptanalysis has little in common since the algorithms and time scales involved in RSA vs. AES operations are very different. Both [23] and [167] contain discussions of countermeasures against the respective attacks, and some of these are also relevant to our attacks (see Section 8.5).

Koeune and Quisquater [110] described a timing attack on a "bad implementation" of AES which uses its algebraic description in a "careless way" (namely, using a conditional branch in the MixColumn operation). That attack is not applicable to common software implementations, but should be taken into account in regard to certain countermeasures against our attack (see Section 8.5.2).

Leakage of memory access information has also been considered in other contexts, yielding theoretical [83] and heuristic [226][225] mitigation methods; these are discussed in Section 8.5.3.

See §10.5 for a discussion of several works following our research.

8.2 Preliminaries

8.2.1 Memory and cache structure

Over the past couple of decades and until recently, CPU speed has been growing at the rate predicted by Moore's law (about 60% per year), while the latency of main memory has been

[1] In our experiments the attack code of [23] failed to get a signal from `dm-crypt` even after a 10 hours run, whereas in an identical setup our Prime+Probe (see §8.3.5) performed full key recovery using 65ms of measurements.

[2] Canteaut et al. [41] claim that their attack exploits only collision effects due to microarchitectural details (i.e., low address bits) and that Bernstein's attack [23] exploits only cache misses (i.e., higher address bits). However, experimentally both attacks yield key bits of both types, as can be expected: the analysis method of [23] also detects collision effects (albeit with lower sensitivity), while the attack setting of [41] inadvertently also triggers systematic cache misses (e.g., due to the encryption function's stack usage and buffers).

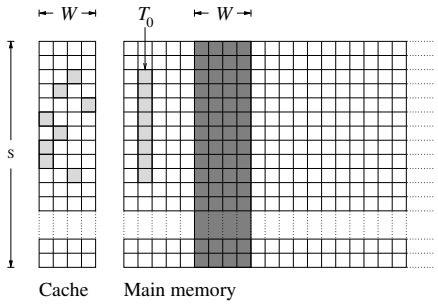

Figure 8.1: Schematic of a set-associative cache. Main memory addresses increase in row-first order. The light gray blocks represent a cached AES lookup table. The dark gray blocks represent the attacker's memory.

decreasing at a much slower rate (7%–9%).[3] Consequentally, a large gap has developed between the two. Complex multi-level cache architectures are employed to bridge this gap, but it still shows through during cache misses: on a typical modern processor, accessing data in the innermost (L1) cache typically requires amortized time on the order of 0.3ns, while accessing main memory may stall computation for 50 to 150ns, i.e., a slowdown of 2–3 orders of magnitude. The cache architectures are optimized to minimize the number of cache misses for typical access patterns, but can be easily manipulated adversarially; to do so we will exploit the special structure in the association between main memory and cache memory.

Modern processors use one or more levels of *set-associative memory cache*. Such a cache consists of storage cells called *cache lines*, each consisting of B bytes. The cache is organized into S *cache sets*, each containing W cache lines[4], so overall the cache contains $B \cdot S \cdot W$ bytes. The mapping of memory addresses into the cache is limited as follows. First, the cache holds copies of aligned blocks of B bytes in main memory (i.e., blocks whose starting address is 0 modulo B), which we will term *memory blocks*. When a cache miss occurs, a full memory block is copied into one of the cache lines. Second, each memory block may be cached only in a specific cache set; specifically, the memory block starting at address a can be cached only in the W cache lines belonging to cache set $\lfloor a/B \rfloor \bmod S$. See Figure 8.1. Thus, the memory blocks are partitioned into S classes, where the blocks in each class contend for the cache lines in a single cache set.

Modern processors have up to 3 levels of memory cache, denoted L1 to L3, with "L1" being the smallest and fastest cache and subsequent levels increasing in size and latency. For simplicity, in the following we mostly ignore this distinction; one has a choice of which cache to exploit, and our experimental attacks used both L1 and L2 effects. Additional complications are discussed in Section 8.3.6. Typical cache parameters are given in Table 8.1.

[3]This relatively slow reduction in DRAM latency has proven so reliable, and founded in basic technological hurdles, that it has been proposed Abadi et al. [1] and Dwork et al. [57] as a basis for proof-of-work protocols.

[4]In common terminology, W is called the *associativity* and the cache is called *W-way set associative*.

CPU model	Level	B (cache line size)	S (cache sets)	W (associativity)	$B \cdot S \cdot W$ (total size)
Athlon 64 / Opteron	L1	64B	512	2	64KB
Athlon 64 / Opteron	L2	64B	1024	16	1024KB
Pentium 4E	L1	64B	32	8	16KB
Pentium 4E	L2	128B	1024	8	1024KB
PowerPC 970	L1	128B	128	2	32KB
PowerPC 970	L2	128B	512	8	512KB
UltraSPARC T1	L1	16B	128	4	8KB
UltraSPARC T1	L2	64B	4096	12	3072KB

Table 8.1: Data cache parameters for popular CPU models

8.2.2 Memory access in AES implementations

This chapter focuses on AES since its memory access patterns are particularly susceptible to cryptanalysis (see Section 10.2.1 for a discussion of other ciphers). The cipher is abstractly defined by algebraic operations and could, in principle, be implemented using just logical and arithmetic operations.[5] However, performance-oriented software implementations on 32-bit (or higher) processors typically use the following alternative formulation, as prescribed in the Rijndael AES submission [52].[6]

Several lookup tables are precomputed once by the programmer or during system initialization. There are 8 such tables, T_0, T_3, T_2, T_3 and $T_0^{(10)}, T_1^{(10)}, T_2^{(10)}, T_3^{(10)}$, each containing 256 4-byte words. The contents of the tables, defined in [52], are inconsequential for most of our attacks.

During key setup, a given 16-byte secret key $\vec{k} = (k_0, \ldots, k_{15})$ is expanded into 10 round keys[7], $\vec{K}^{(r)}$ for $r = 1, \ldots, 10$. Each round key is divided into 4 words of 4 bytes each: $\vec{K}^{(r)} = (K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)})$. The 0-th round key is just the raw key: $K_j^{(0)} = (k_{4j}, k_{4j+1}, k_{4j+2}, k_{4j+3})$ for $j = 0,1,2,3$. The details of the rest of the key expansion are mostly inconsequential.

Given a 16-byte plaintext $\vec{p} = (p_0, \ldots, p_{15})$, encryption proceeds by computing a 16-byte intermediate state $\vec{x}^{(r)} = (x_0^{(r)}, \ldots, x_{15}^{(r)})$ at each round r. The initial state $\vec{x}^{(0)}$ is computed by $x_i^{(0)} = p_i \oplus k_i$ $(i = 0, \ldots, 15)$. Then, the first 9 rounds are computed by updating the intermediate state as follows, for $r = 0, \ldots, 8$:

$$\begin{aligned}
(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}) &\leftarrow T_0[x_0^{(r)}] \oplus T_1[x_5^{(r)}] \oplus T_2[x_{10}^{(r)}] \oplus T_3[x_{15}^{(r)}] \oplus K_0^{(r+1)} \\
(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}) &\leftarrow T_0[x_4^{(r)}] \oplus T_1[x_9^{(r)}] \oplus T_2[x_{14}^{(r)}] \oplus T_3[x_3^{(r)}] \oplus K_1^{(r+1)} \\
(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}) &\leftarrow T_0[x_8^{(r)}] \oplus T_1[x_{13}^{(r)}] \oplus T_2[x_2^{(r)}] \oplus T_3[x_7^{(r)}] \oplus K_2^{(r+1)} \\
(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}) &\leftarrow T_0[x_{12}^{(r)}] \oplus T_1[x_1^{(r)}] \oplus T_2[x_6^{(r)}] \oplus T_3[x_{11}^{(r)}] \oplus K_3^{(r+1)}
\end{aligned} \tag{8.1}$$

[5] Such an implementation would be immune to our attack, but exhibit low performance. A major reason for the choice of Rijndael in the AES competition was the high performance of the implementation analyzed here.

[6] Some software implementations use variants of this formulation with different table layouts; see 8.5.2 for a discussion. The most common variant employs a single table for the last round; most of our attacks analyze only the first few rounds, and are thus unaffected.

[7] We consider AES with 128-bit keys. The attacks can be adapted to longer keys.

Finally, to compute the last round (8.1) is repeated with $r = 9$, except that $T_0, \ldots, T_3$ is replaced by $T_0^{(10)}, \ldots, T_3^{(10)}$. The resulting $\vec{x}^{(10)}$ is the ciphertext. Compared to the algebraic formulation of AES, here the lookup tables represent the combination of ShiftRows, MixColumns and SubBytes operations; the change of lookup tables in the last round is due to the absence of MixColumns.

8.2.3 Notation

We treat bytes interchangeably as integers in $\{0, \ldots, 255\}$ and as elements of $\{0,1\}^8$ that can be XORed. Let δ denote the cache line size B divided by the size of each table entry (usually 4 bytes[8]); on most platforms of interest we have $\delta = 16$. For a byte y and table T_ℓ, we will denote $\langle y \rangle = \lfloor y/\delta \rfloor$ and call this *the memory block of* y in T_ℓ. The significance of this notation is as follows: two bytes y, z fulfill $\langle y \rangle = \langle z \rangle$ iff, when used as lookup indices into the same table T_ℓ, they would cause access to the same memory block[9]; they would therefore be impossible to distinguish based only on a single memory access. For a byte y and table T_ℓ, we say that an AES encryption *accesses the memory block of* y *in* T_ℓ if, according to the above description of AES, at some point during that encryption there is some table lookup of $T_\ell[z]$ where $\langle z \rangle = \langle y \rangle$.

In Section 8.3 we will show methods for discovering (and taking advantage of the discovery) whether the encryption code, invoked as a black box, accesses a given memory block. To this end we define the following predicate: $Q_{\vec{k}}(\vec{p}, \ell, y) = 1$ iff the AES encryption of the plaintext $\vec{p}$ under the encryption key $\vec{k}$ accesses the memory block of index y in T_ℓ at least once throughout the 10 rounds.

Also in Section 8.3, our measurement procedures will sample a *measurement score* from a distribution $M_{\vec{k}}(\vec{p}, \ell, y)$ over $\mathbb{R}$. The exact definition of $M_{\vec{k}}(\vec{p}, \ell, y)$ will vary, but it will approximate $Q_{\vec{k}}(\vec{p}, \ell, y)$ in the following rough sense: for a large fraction of the keys $\vec{k}$, all[10] tables ℓ and a large fraction of the indices $\vec{x}$, for random plaintexts and measurement noise, the expectation of $M_{\vec{k}}(\vec{p}, \ell, y)$ is larger when $Q_{\vec{k}}(\vec{p}, \ell, y) = 1$ than when $Q_{\vec{k}}(\vec{p}, \ell, y) = 0$.

8.3 Synchronous known-data attacks

8.3.1 Overview

The first family of attacks, termed *synchronous attacks*, is applicable in scenarios where the plaintext or ciphertext is known and the attacker can operate synchronously with the encryption

[8] One exception is OpenSSL 0.9.7g on x86-64, which uses 8-byte table entries. The reduced δ improves our attacks.

[9] We assume that the tables are aligned on memory block boundaries, which is usually the case. Non-aligned tables would *benefit* our attacks by leaking an extra bit per key byte in the first round. We also assume for simplicity that all tables are mapped into distinct cache sets; this holds with high probability on many systems (and our practical attacks can handle some exceptions).

[10] This will be relaxed in Section 8.3.7.

on the same processor, by using (or eavesdropping upon) some interface that triggers encryption under an unknown key. For example, a Virtual Private Network (VPN) may allow an unprivileged user to send data packets through a secure channel which uses the same secret key to encrypt all packets. This lets the user trigger encryption of plaintexts that are mostly known (up to some uncertainties in the packet headers), and our attack would thus, under some circumstances, enable any such user to discover the key used by the VPN to protect the packets of other users. As another example, consider the Linux `dm-crypt` and `cryptoloop` services. These allow the administrator to create a virtual device which provides encrypted storage into an underlying physical device, and typically a normal filesystem is mounted on top of the virtual device. If a user has write permissions to *any* file on that filesystem, he can use it to trigger encryptions of known plaintext, and using our attack he is subsequently able to discover the universal encryption key used for the underlying device. We have experimentally demonstrated the latter attack, and showed it to reliably extract the full AES key using about 65ms of measurements (involving just 800 write operations) followed by 3 seconds of analysis.

The attack consists of two stages. In the on-line stage, we obtain a set of random samples, each consisting of a known plaintext and the memory-access side-channel information gleaned during the encryption of that plaintext. This data is cryptanalyzed in an off-line stage, through hypothesis testing: we guess small parts of the key, use the guess to predict some memory accesses, and check whether the predictions are consistent with the collected data. In the following we first describe the cryptanalysis in an idealized form using the predicate Q, and adapt it to the noisy measurements of M. We then show two different methods for obtaining these measurements, detail some experimental results and outline possible variants and extensions.

8.3.2 One-round attack

Our simplest synchronous attack exploits the fact that in the first round, the accessed table indices are simply $x_i^{(0)} = p_i \oplus k_i$ for all $i = 0, \ldots, 15$. Thus, given knowledge of the plaintext byte p_i, any information on the accessed index $x_i^{(0)}$ directly translates to information on key byte k_i. The basic attack, in idealized form, is as follows.

Suppose that we obtain samples of the ideal predicate $Q_{\vec{k}}(\vec{p}, \ell, y)$ for some table ℓ, arbitrary table indices y and known but random plaintexts $\vec{p}$. Let k_i be a key byte such that the first encryption round performs the access "$T_\ell[x_i^{(0)}]$" in (8.1), i.e., such that $i \equiv \ell \pmod 4$. Then we can discover the partial information $\langle k_i \rangle$ about k_i, by testing candidate values $\tilde{k}_i$ and checking them as follows. Consider the samples that fulfill $\langle y \rangle = \left\langle p_i \oplus \tilde{k}_i \right\rangle$. These samples will be said to be *useful for* $\tilde{k}_i$, and we can reason about them as follows. If we correctly guessed $\langle k_i \rangle = \left\langle \tilde{k}_i \right\rangle$ then $Q_{\vec{k}}(\vec{p}, \ell, y) = 1$ for useful samples, since the table lookup "$T_\ell[x_i^{(0)}]$" in (8.1) will certainly access the memory block of y in T_ℓ. Conversely, if $\langle k_i \rangle \neq \left\langle \tilde{k}_i \right\rangle$ then we are assured that "$T_\ell[x_i^{(0)}]$" will *not* access the memory block of y during the first round; however, during the full encryption process there is a total of 36 accesses to T_ℓ (4 in each of the first 9 AES rounds). The remaining 35 accesses

are affected also by other plaintext bytes, so (for sufficiently random plaintexts and avalanche effect) the probability that the encryption will not access that memory block in any round is $(1 - \delta/256)^{35}$. By definition, that is also the probability of $Q_{\vec{k}}(\vec{p}, \ell, y) = 0$, and in the common case $\delta = 16$ it is approximately 0.104.

Thus, after receiving a few dozen useful samples we can identify a correct $\left\langle \tilde{k}_i \right\rangle$ — namely, the one for which $Q_{\vec{k}}(\vec{p}, \ell, y) = 1$ whenever $\langle y \rangle = \left\langle p_i \oplus \tilde{k}_i \right\rangle$. Applying this test to each key byte k_i separately, we can thus determine the top $\log_2(256/\delta) = 4$ bits of every key byte k_i (when $\delta = 16$), i.e., half of the AES key. Note that this is the maximal amount of information that can be extracted from the memory lookups of the first round, since they are independent and each access can be distinguished only up to the size of a memory block.

In reality, we do not have the luxury of the ideal predicate, and have to deal with measurement score distributions $M_{\vec{k}}(\vec{p}, \ell, y)$ that are correlated with the ideal predicate but contain a lot of (possibly structured) noise. For example, we will see that $M_{\vec{k}}(\vec{p}, \ell, y)$ is often correlated with the ideal $Q_{\vec{k}}(\vec{p}, \ell, y)$ for some ℓ but is uncorrelated for others (see Figure 8.5). We thus proceed by averaging over many samples. As done above, we concentrate on a specific key x_i and a corresponding table ℓ. Our measurement will yield samples of the form (p, y, m) consisting of arbitrary table indices y, random plaintexts $\vec{p}$, and measurement scores m drawn from $M_{\vec{k}}(\vec{p}, \ell, y)$. For a candidate key value $\tilde{k}_i$ we define the *candidate score of* $\tilde{k}_i$ as the expected value of m over the samples useful to $\tilde{k}_i$ (i.e., conditioned on $y = p_i \oplus \tilde{k}_i$). We estimate the candidate score by taking the average of m over the samples useful for $\tilde{k}_i$. Since $M_{\vec{k}}(\vec{p}, \ell, y)$ approximates $Q_{\vec{k}}(\vec{p}, \ell, y)$, the candidate score should be noticeably higher when $\left\langle \tilde{k}_i \right\rangle = \langle k_i \rangle$ than otherwise, allowing us to identify the value of k_i up to a memory block.

Indeed, on a variety of systems we have seen this attack reliably obtaining the top nibble of every key byte. Figure 8.2 shows the candidate scores in one of these experiments (see Sections 8.3.5 and 8.3.7 for details); the $\delta = 16$ key byte candidates $\tilde{k}_i$ fulfilling $\left\langle \tilde{k}_i \right\rangle = \langle k_i \rangle$ are easily distinguished.

8.3.3 Two-rounds attack

The above attack narrows each key byte down to one of δ possibilities, but the table lookups in the first AES round can not reveal further information. For the common case $\delta = 16$, the key has 64 remaining unknown bits — still too much for exhaustive search. We thus proceed to analyze the 2nd AES round, exploiting the non-linear mixing in the cipher to reveal additional information. Specifically, we exploit the following equations, easily derived from the Rijndael

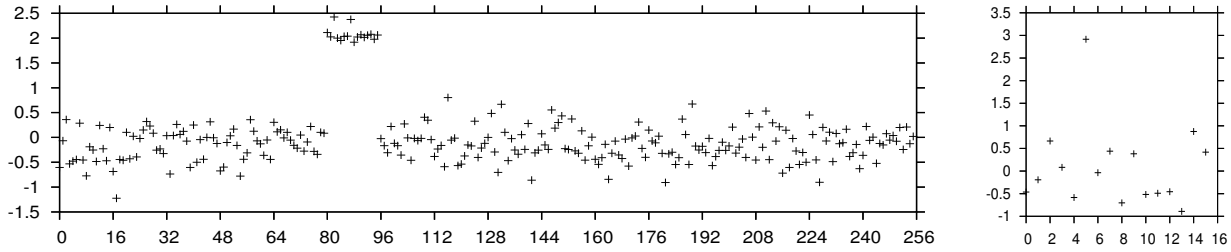

Figure 8.2: Candidate scores for a synchronous attack using Prime+Probe measurements (see §8.3.5), analyzing a `dm-crypt` encrypted filesystem on Linux 2.6.11 running on an Athlon 64, after analysis of 30,000 (left) or 800 (right) triggered encryptions. The horizontal axis is $\tilde{k_5} = p_5 \oplus y$ (left) or $\left\langle \tilde{k}_5 \right\rangle$ (right) and the vertical axis is the average measurement score over the samples fulfilling $y = p_5 \oplus \tilde{k}_5$ (in units of clock cycles). The high nibble of $k_5 = 0\text{x}50$ is easily gleaned.

specification [52], which give the indices used in four of the table lookups in the 2nd round:[11]

$$x_2^{(1)} = s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2 \bullet s(p_{10} \oplus k_{10}) \oplus 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2 \tag{8.2}$$

$$x_5^{(1)} = s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \oplus k_1 \oplus k_5$$

$$x_8^{(1)} = 2 \bullet (p_8 \oplus k_8) \oplus 3 \bullet s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{13}) \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1$$

$$x_{15}^{(1)} = 3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus 2 \bullet s(p_{11} \oplus k_{11}) \oplus s(k_{12}) \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11}$$

Here, $s(\cdot)$ denotes the Rijndael S-box function and $\bullet$ denotes multiplication over GF(256).[12]

Consider, for example, equation (8.2) above, and suppose that we obtain samples of the ideal predicate $Q_{\vec{k}}(\vec{p}, \ell, y)$ for table $\ell = 2$, arbitrary table indices y and known but random plaintexts $\vec{p}$. We already know $\langle k_0 \rangle, \langle k_5 \rangle, \langle k_{10} \rangle, \langle k_{15} \rangle$ and $\langle k_2 \rangle$ from attacking the first round, and we also know the plaintext. The unknown low bits of k_2 (i.e., $k_2 \bmod \delta$), affect only the low bits of $x_2^{(1)}$, (i.e., $x_2^{(1)} \bmod \delta$), and these do not affect which memory block is accessed by "$T_2[x_2^{(1)}]$". Thus, the only unknown bits affecting the memory block accessed by "$T_2[x_2^{(1)}]$" in (8.1) are the lower $\log_2 \delta$ bits of k_0, k_5, k_{10} and k_{15}. This gives a total of δ^4 (i.e., 2^{16} for $\delta = 2^4$) possibilities for candidate values $\tilde{k}_0, \tilde{k}_5, \tilde{k}_{10}, \tilde{k}_{15}$, which can be easily enumerated. To complete the recovery of these four key bytes, we can identify the correct candidate as follows.

Identification of a correct guess is done by a generalization of the hypothesis-testing method used for the one-round attack. For each candidate guess, and each sample, $Q_{\vec{k}}(\vec{p}, \ell, y)$ we evaluate (8.2) using the candidates $\tilde{k}_0, \tilde{k}_5, \tilde{k}_{10}, \tilde{k}_{15}$ while fixing the unknown low bits of k_2 to an arbitrary value. We obtain a predicted index $\tilde{x}_2^{(1)}$. If $\langle y \rangle = \left\langle \tilde{x}_2^{(1)} \right\rangle$ then we say that this sample is *useful* for this candidate, and reason as follows.

[11] These four equations are special in that they involve just 4 unknown quantities, as shown below.

[12] The only property of these functions that we exploit is the fact that $s(\cdot)$, $2 \bullet s(\cdot)$ and $3 \bullet s(\cdot)$ are "random-looking" in a sense specified below.

If the guess was correct then $\langle y \rangle = \left\langle \tilde{x}_2^{(1)} \right\rangle = \left\langle x_2^{(1)} \right\rangle$ and thus "$T_2[x_2^{(1)}]$" certainly causes an access to the memory block of y in T_2, whence $Q_{\vec{k}}(\vec{p}, \ell, y) = 1$ by definition. Otherwise we have $k_i \neq \tilde{k}_i$ for some $i \in \{0,5,10,15\}$ and thus

$$x_2^{(1)} \oplus \tilde{x}_2^{(1)} = c \bullet s(p_i \oplus k_i) \oplus c \bullet s(p_i \oplus \tilde{k}_i) \oplus \cdots$$

for some $c \in \{1,2,3\}$, and since $\vec{p}$ is random the remaining terms are independent of the first two. But for these specific functions the above is distributed close to uniformly. Specifically, it is readily computationally verified, from the definition of AES [52], that the following differential property (cf. [28]) holds: for any $k_i \neq \tilde{k}_i$, $c \in \{1,2,3\}$, $\delta \geq 4$ and $z \in \{0, \ldots, 256/\delta\}$ we always have

$$\Pr_{\vec{p}} \left[\left\langle c \bullet s(p_i \oplus k_i) \oplus c \bullet s(p_i \oplus \tilde{k}_i) \right\rangle \neq z \right] > 1 - (1 - \delta/256)^3 \ .$$

Thus, the probability that "$T_2[x_2^{(1)}]$" in (8.1) does not cause an access to the memory block of y in T_2 is at least $(1 - \delta/256)^3$, and each of the other 35 accesses to T_2 performed during the encryption will access the memory block of y in T_2 with probability $\delta/256$. Hence, $Q_{\vec{k}}(\vec{p}, \ell, y) = 0$ with probability greater than $(1 - \delta/256)^{3+35}$.

We see that each sample eliminates, on average, a $(\delta/256) \cdot (1 - \delta/256)^{38}$-fraction of the candidates — this is the probability, for a wrong candidate, that a random sample is useful for that candidate (i.e., yields a testable prediction) and moreover eliminates that candidate (by failing the prediction). Thus, to eliminate all the wrong candidates out of the δ^4, we need about $\log \delta^{-4} / \log \left(1 - \delta/256 \cdot (1 - \delta/256)^{38}\right)$ samples, i.e., about 2056 samples when $\delta = 16$. Note that with some of our measurement methods the attack requires only a few hundred encryptions, since each encryption can provide samples for multiple y.

Similarly, each of the other three equations above lets us guess the low bits of four distinct key bytes, so taken together they reveal the full key. While we cannot reuse samples between equations since they refer to different tables ℓ, we can reuse samples between the analysis of the first and second round. Thus, if we had access to the ideal predicate Q we would need a total of about 8220 encryptions of random plaintexts, and an analysis complexity of $4 \cdot 2^{16} \cdot 2056 \approx 2^{29}$ simple tests, to extract the full AES key.

In reality we get only measurement scores from the distributions $M_{\vec{k}}(\vec{p}, \ell, y)$ that approximate the ideal predicate $Q_{\vec{k}}(\vec{p}, \ell, y)$. Similarly to the one-round attack, we proceed by computing, for each candidate $\tilde{k}_i$, a candidate score obtained by averaging the measurement scores of all samples useful to $\tilde{k}_i$. We then pick the $\tilde{k}_i$ having the largest measurement score. The number of samples required to reliably obtain all key bytes by this method is, in some experimentally verified settings, only about 7 times larger than the ideal (see Section 8.3.7).

8.3.4 Measurement via Evict+Time

One method for extracting measurement scores is to manipulate the state of the cache before each encryption, and observe the execution time of the subsequent encryption. Recall that we

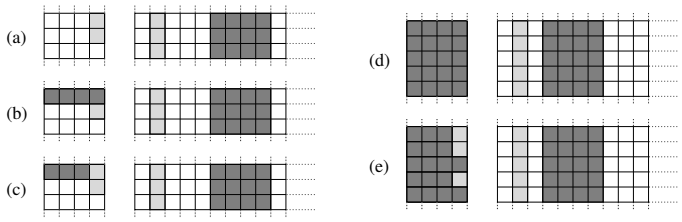

Figure 8.3: Schematics of cache states, in the notation of Figure 8.1. States (a)-(c) depict Evict+Time and (d)-(e) depict Prime+Probe.

assume the ability to trigger an encryption and know when it has begun and ended. We also assume knowledge of the memory address of each table T_ℓ, and hence of the cache sets to which it is mapped.[13] We denote these (virtual) memory addresses by $V(T_\ell)$. In a chosen-plaintext setting, the measurement routine proceeds as follows given a table ℓ, index y into ℓ and plaintext $\vec{p}$:

(a) Trigger an encryption of $\vec{p}$.

(b) *(evict)* Access some W memory addresses, at least B bytes apart, that are all congruent to $V(T_\ell) + y \cdot B/\delta$ modulo $S \cdot B$.

(c) *(time)* Trigger a second encryption of $\vec{p}$ and time it.[14] This is the measurement score.

The rationale for this procedure is as follows. Step (a) makes it highly likely that all table memory blocks accessed during the encryption of $\vec{p}$ are cached[15]; this is illustrated in Figure 8.3(a). Step (b) then accesses memory blocks, in the attacker's own memory space, that happen to be mapped to the same cache set as the memory block of y in T_ℓ. Since it is accessing W such blocks in a cache with associativity W, we expect these blocks to completely replace the prior contents of the cache. Specifically, the memory block of index y in the encryption table T_ℓ is now not in cache; see Figure 8.3(b). When we time the duration of the encryption in (c), there are two possibilities. If $Q_{\vec{k}}(\vec{p}, \ell, y) = 1$, that is if the encryption of the plaintext $\vec{p}$ under the unknown encryption key $\vec{k}$ accesses the memory block of index y in T_ℓ, then this memory block will have to be re-fetched from memory into the cache, leading to Figure 8.3(c). This fetching will slow down the encryption. Conversely, if $Q_{\vec{k}}(\vec{p}, \ell, y) = 0$ then this memory fetch will not occur. Thus, all other things being equal, the expected encryption time is larger when $Q_{\vec{k}}(\vec{p}, \ell, y) = 1$. The gap is on the order of the timing difference between a cache hit and a cache miss.

Figure 8.4 demonstrates experimental results. The bright diagonal corresponds to samples where $\langle y \rangle \oplus \langle p_0 \rangle = \langle k_0 \rangle = 0$, for which the encryption in step (c) always suffers a cache miss.

[13] Also, as before, the cache sets of all tables are assumed to be distinct. See Section 8.3.6 for a discussion of possible complications and their resolution.

[14] To obtain high-resolution timing we use the CPU cycle counter (e.g., on x86 the RDTSC instruction returns the number of clock cycles since the last CPU reset).

[15] Unless the triggered encryption code has excessive internal cache contention, or an external process interfered.

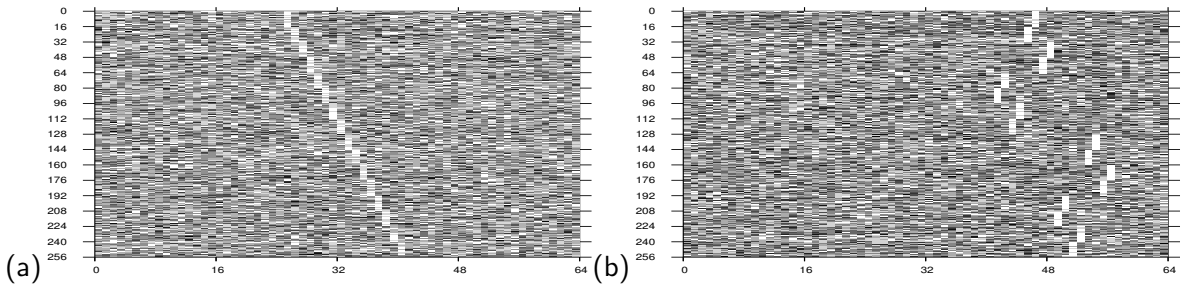

Figure 8.4: Timings (lighter is slower) in Evict+Time measurements on a 2GHz Athlon 64, after 10,000 samples, attacking a procedure that executes an encryption using OpenSSL 0.9.8. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location) and the vertical axis is p_0 (left) or p_5 (right). The patterns of bright areas reveal high nibble values of 0 and 5 for the corresponding key byte values, which are XORed with p_0.

This measurement method is easily extended to a case where the attacker can trigger encryption with plaintexts that are known but not chosen (e.g., by sending network packets to which an uncontrolled but guessable header is added). This is done by replacing step (a) above with one that simply triggers encryptions of arbitrary plaintexts in order to cause *all* table elements to be loaded into cache. Then the measurement and its analysis proceed as before.

The weakness of this measurement method is that, since it relies on timing the triggered encryption operation, it is very sensitive to variations in the operation. In particular, triggering the encryption (e.g., through a kernel system call) typically executes additional code, and thus the timing may include considerable noise due to sources such as instruction scheduling, conditional branches, page table misses, and other sources cache contention. Indeed, using this measurement method we were able to extract full AES keys from an artificial service doing AES encryptions using OpenSSL library calls[16], but not from more typical "heavyweight" services. For the latter, we invoked the alternative measurement method described in the next section.

8.3.5 Measurement via Prime+Probe

This measurement method tries to discover the set of memory blocks read by the encryption *a posteriori*, by examining the state of the cache after encryption. This method proceeds as follows. The attacker allocates a contiguous byte array $A[0, \ldots, S \cdot W \cdot B - 1]$, with start address congruent modulo $S \cdot B$ to the start address of T_0.[17] Then, given a plaintext $\vec{p}$, it obtains measurement scores for all tables ℓ and all indices y and does so using a *single* encryption:

(a) *(prime)* Read a value from every memory block in A.

(b) Trigger an encryption of $\vec{p}$.

[16]For this artificial scenario, [23] also demonstrated key extraction.

[17]For simplicity, here we assume this address is known, and that T_0, T_1, T_2, T_3 are contiguous.

(c) *(probe)* For every table $\ell = 0, \ldots 3$ and index $y = 0, \delta, 2\delta, \ldots, 256 - \delta$:

- Read the W memory addresses $A[1024\ell + 4y + tSB]$ for $t = 0, \ldots, W - 1$. The total time it takes to perform these W memory accesses is the measurement score for ℓ and y, i.e., our sample of $M_{\vec{k}}(\vec{p}, \ell, y)$.[18]

Step (a) completely fills the cache with the attacker's data; see Figure 8.3(e). The encryption in step (b) causes partial eviction; see Figure 8.3(f). Step (c) checks, for each cache set, whether the attacker's data is still present after the encryption: cache sets that were accessed by the encryption in step (b) will incur cache misses in step (c), but cache sets that were untouched by the encryption will not, and thus induces a timing difference.

Crucially, the attacker is timing a simple operation performed by *itself*, as opposed to a complex encryption service with various unknown overheads executed by someone else (as in the Evict+Time approach); this is considerably less sensitive to timing variance, and oblivious to time randomization or canonization (which are frequently proposed countermeasures against timing attacks; see Section 8.5). Another benefit lies in inspecting all cache sets simultaneously after each encryption, so that each encryption effectively yields $4 \cdot 256/\delta$ samples of measurement score, rather than a single sample.

An example of the measurement scores obtained by this method, for a real cryptographic system, are shown in Figure 8.5. Note that to obtain a visible signal it is necessary to normalize the measurement scores by subtracting, from each sample, the average timing of its cache set. This is because different cache sets are affected differently by auxiliary memory accesses (e.g., variables on the stack and I/O buffers) during the system call. These extra accesses depend on the inspected cache set but are nearly independent of the plaintext byte; thus they affect each column uniformly and can be subtracted away. Major interruptions, such as context switches to other processes, are filtered out by excluding excessively long time measurements.

8.3.6 Practical complications

Above we have ignored several potential complications. One of these is that the attacker does not know where the victim's lookup tables reside in memory. It may be hard to tell in advance, or it might be randomized by the victim.[19] However, the attacker usually does know the layout (up to unknown global offset) of the victim's lookup tables, and this enables the following simple procedure: try each possible table offset in turn, and apply the one-round attack assuming this offset. Then pick the offset that gave the maximal candidate score. In our experiments this method works very well, even on a real, noisy system (see Figure 8.6). Often, it even suffices to simply look for a frequently-accessed range of memory of the right size (see Figure 8.7).

[18] We perform probing using pointer-chasing to ensure loads which are not reorderable by the CPU's internal optimizations. To avoid "polluting" our samples, the probe code stores each obtained sample into the same cache set it measured. On some platforms one can improve the timing gap by using writes instead of reads, or more than W reads.

[19] For example, recent Linux kernels randomize memory offsets.

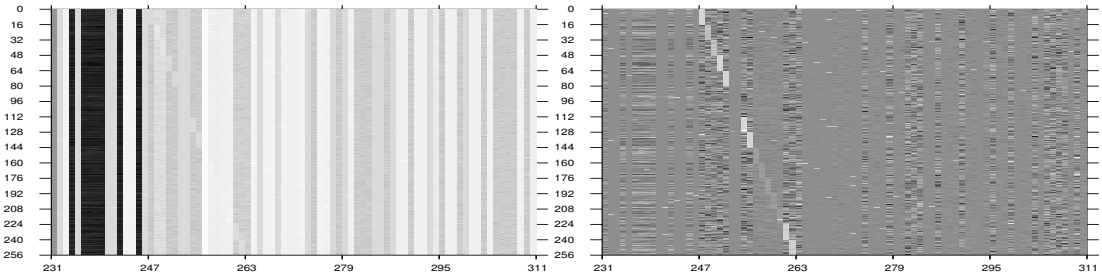

Figure 8.5: Prime+Probe attack using 30,000 encryption calls on a 2GHz Athlon 64, attacking Linux 2.6.11 `dm-crypt`. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location) and the vertical axis is p_0. Left: raw timings (lighter is slower). Right: after subtraction of the average timing of each cache set (i.e., column). The bright diagonal reveals the high nibble of $p_0 = 0\text{x}00$.

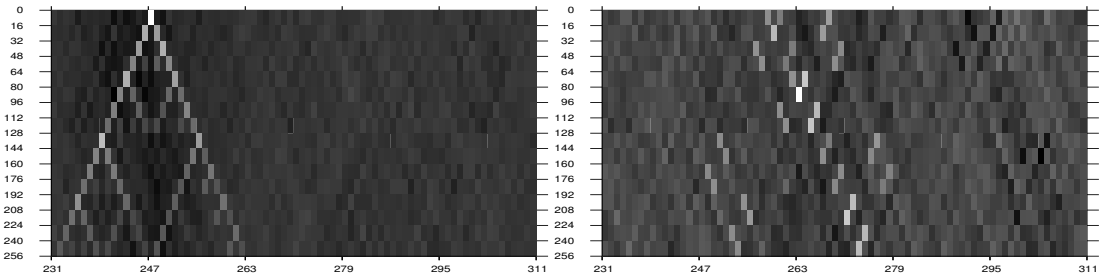

Figure 8.6: Scores (lighter is higher) for combinations of key byte candidate (vertical axis) and table offset candidate (horizontal axis). The correct combinations are clearly identified as the bright spots at the head of the Sierpinski-like triangle (which is row-permuted on the right). Note the correct relative offsets of tables T_0 (left) and T_1 (right) . This is the same dataset as in Figure 8.5.

Another complication is the distinction between virtual and physical memory addresses. The mapping between the two is done in terms of full *memory pages* (i.e., aligned ranges of addresses). These can be of different sizes, even on a single system, but are usually large enough to contain the all the tables used in the first 9 AES rounds. In the above descriptions, and in some of our attacks, we used the knowledge of both virtual and physical addresses of the victim's tables. Sometimes this is available (e.g., when the attacker and victim use the same shared library); it is also not a concern when the cache uses indexing by virtual address. When attacking a physically indexed cache, the attacker can run a quick preprocessing stage to gain the necessary knowledge about the mapping from virtual to physical addresses, by analysis of cache collisions between pages. Some operating systems perform page coloring [103], which makes this even easier. Alternatively, in both measurement methods, the attacker can increase the number of pages accessed to well above the cache associativity, thereby making it likely that the correct pages are hit; we have verified experimentally that this simple method works, albeit at a large cost in measurement time (a factor of roughly 300).

Additional complications arise, such as methods for obtaining high-resolution, low-latency time measurements. These are all surmountable, but are omitted here for brevity.

8.3.7 Experimental results

We have tested the synchronous attacks against AES in various settings. To have an initial "clean" testing environment for our attack code, we started out using OpenSSL library calls as black-box functions, pretending we have no access to the key. In this setting, and with full knowledge of the relevant virtual and physical address mappings, using Prime+Probe measurements we recover the full 128-bit AES key after only 300 encryptions on Athlon 64, and after 16,000 encryptions on Pentium 4E.[20] In the same setting, but without any knowledge about address mappings (and without any attempt to discover it systematically) we still recover the full key on Athlon 64 after 8,000 encryptions.

We then proceeded to test the attacks on a real-life encrypted filesystem. We set up a Linux `dm-crypt` device, which is a virtual device which encrypts all data at the sector level (here, using 128-bit AES encryptions in ECB mode). The encrypted data is saved in an underlying storage device (here, a loopback device connected to a regular file). On top of the `dm-crypt` device, we created and mounted an ordinary ext2 filesystem. We triggered encryptions by performing writes to an ordinary file inside that file system, after opening it in `O_DIRECT` mode; each write consisted of a random 16-byte string repeated 32 times. Running this on the Athlon 64 with knowledge about address mappings, we succeeded in extracting the full key after just 800 write operations done in 65ms (including the analysis of the cache state after each write), followed by 3 seconds of off-line analysis. Data from two analysis stages for this kind of attack are shown in Figures 8.5 and 8.6 (the figures depict a larger number of samples, in order to make the results evident not only to sensitive statistical tests but even to cursory visual inspection).

The Evict+Time measurements (Figure 8.4) are noisier, as expected, but still allow us to recover the secret key using about 500,000 samples when attacking OpenSSL on Athlon 64. Gathering the data takes about half a minute of continuous measurement, more than three orders of magnitude slower than the attacks based on Prime+Probe.

8.3.8 Variants and extensions

There are many possible extensions to the basic techniques described above. The following are a few notable examples.

So far we have discussed known-plaintext attacks. All of these techniques can be applied analogously in known-ciphertext setting. In fact the latter are significantly more efficient for AES implementations of the form given in Section 8.2.2, since the last round uses a dedicated set of tables and the noise due to other rounds is thus eliminated. Moreover, in the last round we have non-linearity but no MixColumn operation, so we can easily extract the full key without analyzing additional rounds. Note that even in the case of known-plaintext, the final guess of

[20] The Athlon 64 processor yielded very stable timings, whereas the Pentium 4E timings exhibited considerable variance (presumably, due to some undocumented internal state).

the key can be efficiently verified by checking the resulting predictions for the lookups in the last round.[21]

Since AES decryption is very similar to encryption, all of our attacks can be applied to the decryption code just as easily. Moreover, the attacks are also applicable when AES is used in MAC mode, as long as either the input or output of some AES invocations is known.

In the two-round attack, we can guess byte *differences* $\tilde{\Delta} = k_i \oplus k_j$ and consider plaintexts such that $p_i \oplus p_j = \tilde{\Delta}$, in order to cancel out pairs of terms $S(k_i \oplus p_i) \oplus S(k_j \oplus p_j)$ in (8.2). This reduces the complexity of analysis (we guess just $\tilde{\Delta}$ instead of both $\tilde{k_i}$ and $\tilde{k_j}$), at the cost of using more measurements.

To verify the results of the second-round analysis, or in case some of the tables cannot be analyzed due to excessive noise, we can use the other 12 lookups in the second round, or even analyze the third round, by plugging in partial information obtained from good tables.

Typically, loading a memory block into a cache line requires several memory transfer cycles due to the limited bandwidth of the memory interface. Consequently, on some processors the load latency depends on the offset of the address within the loaded memory block. Such variability can leak information on memory accesses with resolution better than δ, hence an analysis of the first round via Evict+Time can yield additional key bits. Cache bank collisions (e.g., in Athlon 64 processors) likewise cause timing to be affected by low address bits.

We believe this attack can be converted into a remote attack on a network-triggerable cryptographic network process (e.g., IPsec [102] or OpenVPN [157]).[22] The cache manipulation can be done remotely, for example, by triggering accesses to the state tables employed by the host's TCP stack, stateful firewall or VPN software. These state tables reside in memory and are accessed whenever a packet belonging to the respective network connection is seen. The attacker can thus probe different cache sets by sending packets along different network connections, and also measure access times by sending packets that trigger a response packet (e.g., an acknowledgment or error). If a large number of new connections is opened simultaneously, the memory addresses of the slots assigned to these connections in the state tables will be strongly related (e.g., contiguous or nearly so), and can be further ascertained by finding slots that are mapped to the same cache set (by sending appropriate probe packets and checking the response time). Once the mapping of the state table slots to cache sets is established, all of the aforementioned attacks can be carried out; however, the signal-to-noise (and thus, the efficiency) of this technique remains to be evaluated.

[21] This was indeed demonstrated by [153] subsequent to our publication; see §10.5.
[22] This is ruled out in [41], though no justification is given.

8.4 Asynchronous attacks

8.4.1 Overview

While the synchronous attack presented in the previous section leads to very efficient key recovery, it is limited to scenarios where the attacker has some interaction with the encryption code which allows him to obtain known plaintexts and execute code just before and just after the encryption. We now proceed to describe a class of attacks that eliminate these prerequisites. The attacker will execute his own program on the same processor as the encryption program, but without any explicit interaction such as inter-process communication or I/O, and the only knowledge assumed is about the non-uniform distribution of the plaintexts or ciphertexts (rather than their specific values). Essentially, the attacker will ascertain patterns of memory access performed by other processes just by performing and measuring accesses to its own memory. This attack is more constrained in the hardware and software platforms to which it applies, but it is very effective on certain platforms, such as the increasingly popular CPU architectures which implement simultaneous multithreading.

8.4.2 One-Round Attack

The basic form of this attack works by obtaining a statistical profile of the frequency of cache set accesses. The means of obtaining this will be discussed in the next section, but for now we assume that for each table T_ℓ and each memory block $n = 0, \ldots, 256/\delta - 1$ we have a *frequency score* value $F_\ell(n) \in \mathbb{R}$, that is strongly correlated with the relative frequencies. For a simple but common case, suppose that the attacker process is performing AES encryption of English text, in which most bytes have their high nibble set to 6 (i.e., lowercase letters `a` through `p`). Since the actual table lookups performed in round 1 of AES are of the form "$T_\ell[x_i^{(0)}]$" where $x_i^{(0)} = p_i \oplus k_i$, the corresponding frequency scores $F_\ell(n)$ will have particularly large values when $n = 6 \oplus \langle k_i \rangle$ (assuming $\delta = 16$). Thus, just by finding the n for which $F_\ell(n)$ is large and XORing them with the constant 6, we get the high nibbles $\langle k_i \rangle$.

Note, however, that we cannot distinguish the order of different memory accesses to the same table, and thus cannot distinguish between key bytes k_i involved in the first-round lookup to the same table ℓ. There are four such key bytes per table (for example, k_0, k_5, k_{10}, k_{15} affect T_0; see Section 8.2.2). Thus, when the four high key nibbles $\langle k_i \rangle$ affecting each table are distinct (which happens with probability $\left((16!/12!)/16^4\right)^4 \approx 0.2$), the above reveals the top nibbles of all key bytes but only up to four disjoint permutations of 4 elements each. Overall this gives $64/\log_2(4!^4) \approx 45.66$ bits of key information, somewhat less than the one-round synchronous attack. When the high key nibbles are not necessarily disjoint we get more information, but the analysis of the signal is somewhat more complex.

More generally, suppose the attacker knows the first-order statistics of the plaintext; these can usually be determined just from the type of data being encrypted (e.g., English text, numerical

data in decimal notation, machine code or database records).[23] Specifically, suppose that the attacker knows $R(n) = \Pr[\langle p_i \rangle = n]$ for $n = 0, \ldots, (256/\delta - 1)$, i.e., the histogram of the plaintext bytes truncated into blocks of size δ (where the probability is over all plaintext blocks and all bytes i inside each block). Then the partial key values $\langle k_i \rangle$ can be identified by finding those that yield maximal correlation between $F_\ell(n)$ and $R(n \oplus \langle k_i \rangle)$.

8.4.3 Measurements

One measurement method exploits the simultaneous multithreading (SMT, called "HyperThreading" in Intel Corp. nomenclature) feature available in some high-performance processors (e.g., most modern Pentium and Xeon processors, as well as POWER5 processors, UltraSPARC T1 and others).[24] This feature allows concurrent execution of multiple processes on the same physical processor, with instruction-level interleaving and parallelism. When the attacker process runs concurrently with its victim, it can analyze the latter's memory accesses in real time and thus obtain higher resolution and precision; in particular, it can gather detailed statistics such as the frequency scores $F_\ell(n) \in \mathbb{R}$. This can be done via a variant of the Prime+Probe measurements of Section 8.3.5, as follows.

For each cache set, the attacker thread runs a loop which closely monitors the time it takes to repeatedly load a set of memory blocks that exactly fills that cache set with W memory blocks (similarly to step (c) of the Prime+Probe measurements).[25] As long as the attacker is alone in using the cache set, all accesses hit the cache and are very fast. However, when the victim thread accesses a memory location which maps to the set being monitored, that causes one of the attacker's cache lines to be evicted from cache and replaced by a cache line from the victim's memory. This leads to one or (most likely) more cache misses for the attacker in subsequent loads, and slows him down until his memory once more occupies all the entries in the set. The attacker thus measures the time over an appropriate number of accesses and computes their average, thus obtaining the frequency score $F_\ell(n)$.

8.4.4 Experimental results

Attacking a series of processes encrypting English text with the same key using OpenSSL, we effectively retrieve 45.7 bits of information[26] about the key after gathering timing data for just 1 minute. Timing data from one of the runs is shown in Figure 8.7.

[23] Note that even compressed data is likely to have strong first-order statistical biases at the beginning of each compressed chunk, especially when file headers are employed.

[24] We stress that this attack can be carried out also in the absence of simultaneous multithreading; see Section 8.4.5.

[25] Due to the time-sensitivity and effects such as prefetching and instruction reordering, getting a significant signal requires a carefully crafted architecture-specific implementation of the measurement code.

[26] For keys with distinct high nibbles in each group of 4; see Section 8.4.1.

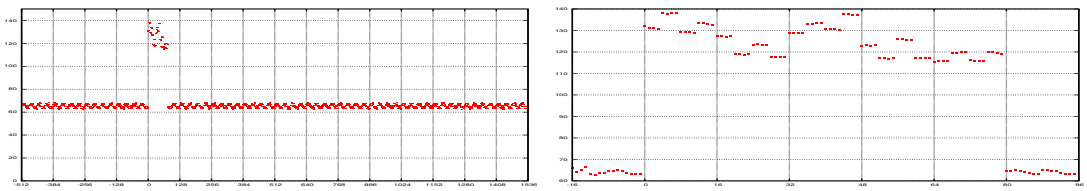

Figure 8.7: Frequency scores for OpenSSL AES encryption of English text. Horizontal axis: cache set. Timings performed on 3GHz Pentium 4E with HyperThreading. To the right we zoom in on the AES lookup tables; the pattern corresponds to the top nibbles of the secret key `0x004080C0105090D02060A0E03070B0F0`.

8.4.5 Variants and extensions

This attack vector is quite powerful, and has numerous possible extensions, such as the following.

The second round can be analyzed using higher-order statistics on the plaintext, yielding enough key bits for exhaustive search.

If measurements can be made to detect the order of accesses (which we believe is possible with appropriately crafted code), the attacker can analyze more rounds as well as extract the unknown permutations from the first round. Moreover, if the temporal resolution suffices to observe adjacent rounds in a single encryption, then it becomes possible to recover the complete key without even knowing the plaintext *distribution*, as long as it is sufficiently nonuniform.

We have demonstrated the attack on a Pentium 4E with HyperThreading, but it can also be performed on other platforms without relying on simultaneous multithreading. The essential requirements is that the attacker can execute its own code while an encryption is in progress, and this can be achieved by exploiting the interrupt mechanism. For example, the attacker can predict RTC or timer interrupts and yield the CPU to the encrypting process a few cycles before such an interrupt; the OS scheduler is invoked during the interrupt, and if dynamic priorities are set up appropriately in advance then the attacker process will regain the CPU and can analyze the state of the cache to see with great accuracy what the encrypting process accessed during those few cycles.[27]

On multi-core processors, the lowest-level caches (L1 and sometimes L2) are usually private to each core; but if the cryptographic code occasionally exceeds these private caches and reaches caches that are shared among the cores (L2 or L3) then the asynchronous attack becomes applicable at the cross-core level. In SMP systems, cache coherency mechanisms may be exploitable for similar effect.

As in the synchronous case, one can envision remote attack variants that take advantage of data structures to which accesses can be triggered and timed through a network (e.g., the TCP state table).

[27] This was indeed subsequently demonstrated by [153]; see §10.5.

8.5 Countermeasures

In the following we discuss several potential methods to mitigate the information leakage. Since these methods have different trade-offs and are architecture- and application-dependent, we cannot recommend a single recipe for all implementors. Rather, we aim to present the realistic alternatives along with their inherent merits and shortcomings. We focus our attention on methods that can be implemented in software, whether by operating system kernels or normal user processes, running under today's common general-purpose processors. Some of these countermeasures are presented as specific to AES, but have analogues for other primitives. Countermeasures which require hardware modification are discussed in [161, 162, 23, 167, 163].

Caveat: due to the complex architecture-dependent considerations involved, we expect the secure implementation of these countermeasures to be a very delicate affair. Implementers should consider all exploitable effects given in [23], and carefully review their architecture for additional effects.

8.5.1 Avoiding memory accesses

Our attacks exploit the effect of memory access on the cache, and would thus be completely mitigated by an implementation that does not perform any table lookups. This may be achieved by the following approaches.

First, one could use an alternative description of the cipher which replaces table lookups by an equivalent series of logical operations. For AES this is particularly elegant, since the lookup tables have concise algebraic descriptions, but performance is degraded by over an order of magnitude.[28]

Another approach is that of bitslice implementations [26]. These employ a description of the cipher in terms of bitwise logical operations, and execute multiple encryptions simultaneously by vectorizing the operations across wide registers. Their performance depends heavily on the structure of the cipher, the processor architecture and the possibility of amortizing the cost across several simultaneous encryptions (which depends on the use of an appropriate encryption mode). For AES, we expect (but have not yet verified) that amortized performance would be comparable to that of a lookup-based implementation, but its relevance is application-dependent.

Finally, one could use lookup tables but place the tables in registers instead of cache. Some architectures (e.g., x86-64 and PowerPC AltiVec) have register files sufficiently large to hold the 256-byte S-box table, but reasonable performance seems unlikely.

8.5.2 Alternative lookup tables

For AES, there are several similar formulations of the encryption and decryption algorithms that use different sets of lookup tables. Above we have considered the most common implementation,

[28] This kind of implementation has also been attacked through the timing variability in some implementations [110].

employing four 1024-byte tables $T_0, \ldots, T_3$ for the main rounds. Variants have been suggested with one 256-byte table (for the S-box), two 256-bytes tables (adding also $2 \bullet S[\cdot]$), one 1024-byte table (just T_0 with the rest obtained by rotations), and one 2048-byte table ($T_0, \ldots, T_3$ compressed into one table with non-aligned lookups). The same applies to the last round tables, $T_0^{(10)}, \ldots, T_3^{(10)}$.

In regard to the synchronous attacks considered in Section 8.3, the effect of using smaller tables is to decrease the probability ρ that a given memory block will not be accessed during the encryption (i.e., $Q_{\vec{k}}(\vec{p}, \ell, y) = 0$) when the candidate guess $\tilde{k}_i$ is wrong. Since these are the events that rule out wrong candidates, the amount of data and analysis in the one-round attack is inversely proportional to $\log(1 - \rho)$.

For the most compact variant with a single 256-byte table, and $\delta = 64$, the probability is $\rho = (1 - 1/4)^{160} \approx 2^{-66.4}$, so the synchronous attack is infeasible – we're unlikely to ever see an unaccessed memory block. For the next most compact variant, using a single 1024 bytes table, the probability is $\rho = (1 - 1/16)^{160} \approx 2^{-14.9}$, compared to $\rho \approx 0.105$ in Section 8.3.2. The attack will thus take about $\log(1 - 0.105)/\log(1 - 2^{-14.9}) \approx 3386$ times more data and analysis, which is inconvenient but certainly feasible for the attacker. The variant with a single 2KB table ($8 \rightarrow 64$ bit) has $\rho = (1 - 1/32)^{160}$, making the synchronous attack just 18 times less efficient than in Section 8.3.2 and thus still doable within seconds.

For asynchronous attacks, if the attacker can sample at intervals on the order of single table lookups (which is architecture-specific) then these alternative representations provide no appreciable security benefit. We conclude that overall, this approach (by itself) is of very limited value. However, it can be combined with some other countermeasures (see Sections 8.5.3, 8.5.5, 8.5.8).

8.5.3 Data-independent memory access pattern

Instead of avoiding table lookups, one could employ them but ensure that the pattern of accesses to the memory is completely independent of the data passing through the algorithm. Most naively, to implement a memory access one can read *all* entries of the relevant table, in fixed order, and use just the one needed. Modern CPUs analyze dependencies and reorder instructions, so care (and overhead) must be taken to ensure that the instruction and access scheduling, and their timing, are completely data-independent.

If the processor leaks information only about whole memory blocks (i.e., not about the low address bits),[29] then it suffices that the sequence of accesses to *memory blocks* (rather than *memory addresses*) is data-independent. To ensure this one can read a representative element from every memory block upon every lookup.[30] For the implementation of AES given in Section 8.2.2 and

[29] This assumption is false for the Athlon 64 processor (due to cache bank collision effects), and possibly for other processors as well. See Section 8.3.8 and [23].

[30] This approach was suggested by Intel Corp. [37] for mitigating the attack of Percival on RSA [167], and incorporated into OpenSSL 0.9.7h. In the case of RSA the overhead is insignificant, since other parts of the computation dominate the running time.

the typical $\delta = 16$, this means each logical table access would involve 16 physical accesses, a major slowdown. Conversely, in the formulation of AES using a single 256-byte table (see Section 8.5.2), the table consists of only 4 memory blocks (for $\delta = 64$), so every logical table access would (the dominant innermost-loop operation) involve just 4 physical accesses; but this formulation of AES is inherently very slow.

A still looser variant is to require only that the sequence of accesses to *cache sets* is data-independent (e.g., store each AES table in memory blocks that map to a single cache set). While this poses a challenge to the cryptanalyst, it does not in general suffice to eliminate the leaked signal: an attacker can still initialize the cache to a state where only a specific memory block is missing from cache, by evicting all memory blocks from the corresponding cache set and then reading back all but one (e.g., by triggering access to these blocks using chosen plaintexts); he can then proceed as in Section 8.3.4. Moreover, statistical correlations between memory block accesses, as exploited in the collision attacks of Tsunoo et al. [206][205], are still present.

Taking a broader theoretical approach, Goldreich and Ostrovsky [83] devised a realization of *Oblivious RAM*: a generic program transformation which hides all information about memory accesses. This transformation is quite satisfactory from an (asymptotic) theoretical perspective, but its concrete overheads in time and memory size are too high for most applications.[31] Moreover, it employs pseudo-random functions, whose typical realizations can also be attacked since they employ the very same cryptographic primitives we are trying to protect.[32]

Xhuang, Zhang, Lee and Pande addressed the same issue from a more practical perspective and proposed techniques based on shuffling memory content whenever it is accessed [226] or occasionally permuting the memory and keeping the cache locked between permutations [225]. Both techniques require non-trivial hardware support in the processor or memory system, and do not provide perfect security in the general case.

A simple heuristic approach is to add noise to the memory access pattern by adding spurious accesses, e.g., by performing a dummy encryption in parallel to the real one. This decreases the signal visible to the attacker (and hence necessitates more samples), but does not eliminate it.

8.5.4 Application-specific algorithmic masking

There is extensive literature about side-channel attacks on hardware ASIC and FPGA implementations, and corresponding countermeasures. Many of these countermeasures are implementation-specific and thus of little relevance to us, but some of them are algorithmic. Of particular interest are masking techniques, which effectively randomize all data-dependent operations by applying random transformations; the difficulty lies, of course, in choosing transformations that can be

[31] The Oblivious RAM model of [83] protects against a stronger adversary which is also able to corrupt the data in memory. If one is interested only in achieving correctness (not secrecy) in the face of such corruption, then Blum et al. [30] provide more efficient schemes and Naor and Rothblum [148] provide strong lower bounds.

[32] In [83] it is assumed that the pseudorandom functions are executed completely within a secure CPU, without memory accesses. If such a CPU was available, we could use it to run the AES algorithm itself.

stripped away after the operation. One can think of this as homomorphic secret sharing, where the shares are the random mask and the masked intermediate values. For AES, several masking techniques have been proposed (see e.g. [160] and the references within). However, these are designed to protect only against first-order analysis, i.e., against attacks that measure some aspect of the state only at one point in the computation. Note that our asynchronous attacks do not fall into this category. Moreover, the security proofs consider leakage only of specific intermediate values, which do not correspond to the ones leaking via memory access metadata. Lastly, every AES masking method we are aware of has either been shown to be insecure even for its original setting (let alone ours), or is significantly slower in software than a bitslice implementation (see Section 8.5.1). Thus, this venue presently seems unfruitful.

8.5.5 Cache state normalization and process blocking

To foil the synchronous attacks of Section 8.3, it suffices to ensure that the cache is at a data-independent normalized state (e.g., by loading all lookup table elements into cache) at any entry to and exit from the encryption code (including interrupt and context switching by the operating system). Thus, to foil the Prime+Probe attack it suffices to normalize the state of the cache after encryption. To foil the Evict+Time attack one needs to normalize the state of the cache immediately before encryption (as in [161]), and also after every interrupt occurring during an encryption (the memory accesses caused by the interrupt handler will affect the state of the cache in some semi-predictable way and can thus be exploited by the attacker similarly to the Evict stage of Evict+Time). Performing the normalization after interrupts typically requires operating system support (see Section 8.5.11). As pointed out in [23, Sections 12 and 14], it should be ensured that the table elements are not evicted by the encryption itself, or by accesses to the stack, inputs or outputs; this is a delicate architecture-dependent affair.

A subtle aspect is that the cache state, which we seek to normalize, includes a hidden state which is used by the CPU's cache eviction algorithm (typically Least Recently Used or variants thereof). If multiple lookup table memory blocks are mapped to the same cache set (e.g., OpenSSL on the Pentium 4E; see Table 8.1), the hidden state could leak information about which of these blocks was accessed last even if all of them are cached; an attacker can exploit this to analyze the last rounds in the encryption (or decryption).

All of these countermeasures provide little protection against the asynchronous attacks of Section 8.4. To fully protect against those, during the encryption one would have to disable interrupts and stop simultaneous threads (and perhaps also other processors on an SMP machine, due to the cache coherency mechanism). This would significantly degrade performance on SMT and SMP machines, and disabling interrupts for long durations will have adverse effects. A method for blocking processes more selectively based on process credentials and priorities is suggested in [167].

Note that normalizing the cache state frequently (e.g., by reloading all tables after every AES round) would merely reduce the signal-to-noise of the asynchronous attacks, not eliminate them.

8.5.6 Disabling cache sharing

To protect against software-based attacks, it would suffice to prevent cache state effects from spanning process boundaries. Alas, practically this is very expensive to achieve. On a single-threaded processor, it would require flushing all caches during every context switch. On a processor with simultaneous multithreading, it would also require the logical processors to use separate logical caches, statically allocated within the physical cache; some modern processors do not support such a mode. One would also need to consider the effect of cache coherency mechanisms in SMP configurations.

A relaxed version would activate the above means only for specific processes, or specific code sections, marked as sensitive. This is especially appropriate for the operating system kernel, but can be extended to user processes as explained in Section 8.5.11.

To separate two processes w.r.t. to the attacks considered here, it suffices[33] to ensure that all memory accessible by one process is mapped into a group of cache sets that is disjoint from that of the other process.[34] In principle, this can be ensured by the operating system virtual memory allocator, through a suitable page coloring algorithm. Alas, this fails on both of the major x86 platforms: in modern Intel processors every 4096-byte memory page is mapped to every cache set in the L1 cache (see Table 8.1), while in AMD processors the L1 cache is indexed by virtual addresses (rather than physical addresses) and these are allocated contiguously.

8.5.7 Static or disabled Cache

One brutal countermeasure against the cache-based attacks is to completely disable the CPU's caching mechanism.[35] Of course, the effect on performance would be devastating, slowing down encryption by a factor of about 100. A more attractive alternative is to activate a "no-fill" mode[36] where the memory accesses are serviced from the cache when they hit it, but accesses that miss the cache are serviced directly from memory (without causing evictions and filling). The encryption routine would then proceed as follows:

(a) Preload the AES tables into cache

(b) Activate "no-fill" mode

(c) Perform encryption

(d) Deactivate "no-fill" mode

[33] In the absence of low-address-bit leakage due to cache bank collisions.

[34] This was proposed to us by Úlfar Erlingsson of Microsoft Research.

[35] Some stateful effects would remain, such as the DRAM bank activation. These might still provide a low-bandwidth side channel in some cases.

[36] Not to be confused with the "disable cache flushing" mode suggested in [162], which is relevant only in the context of smartcards.

The section spanning (a) and (b) is critical, and attacker processes must not be allowed to run during this time. However, once this setup is completed, step (c) can be safely executed. The encryption per se would not be slowed down significantly (assuming its inputs are in cache when "no-fill" is enabled), but its output will not be cached, leading to subsequent cache misses when the output is used (in chaining modes, as well as for the eventual storage or transmission). Other processes executed during (c), via multitasking or simultaneous multithreading, may incur a severe performance penalty. Breaking the encryption chunks into smaller chunks and applying the above routine to each chunk would reduce this effect somewhat, by allowing the cache to be occasionally updated to reflect the changing memory working set.

Intel's family of Pentium and Xeon processors supports such a mode,[37] but the cost of enabling and disabling it are prohibitive. Also, some ARM implementations have a mini-cache which can be locked, but it is too small for the fastest table-based formulations. We do not know which other processors families currently offer this functionality.

This method can be employed only in privileged mode, which is typically available only to the operating system kernel (see Section 8.5.11), and may be competitive performance-wise only for encryption of sufficiently long sequences. In some cases it may be possible to delegate the encryption to a co-processor with the necessary properties. For example, IBM's Cell processor consists of a general-purpose (PowerPC) core along with several "Synergistic Processing Element" (SPE) cores. The latter have a fast local memory but it is not a cache per se, i.e., there are no automatic transfers to or from main memory, thus, SPEs employed as cryptographic co-processors would not be susceptible to this attack.[38]

8.5.8 Dynamic table storage

The cache-based attacks observe memory access patterns to learn about the table lookups. Instead of eliminating these, we may try to decorrelate them. For example, one can use many copies of each table, placed at various offsets in memory, and have each table lookup (or small group of lookups) use a pseudorandomly chosen table. Ideally, the implementation will use S copies of the tables, where S is the number of cache sets (in the largest relevant cache). However, this means most table lookups will incur cache misses. Somewhat more compactly, one can use a single table, but pseudorandomly move it around memory several times during each encryption.[39] If the tables reside in different memory pages, one should consider and prevent leakage (and performance degradation) through page table cache (i.e., Table Lookaside Buffer) misses.

[37] Enable the CD bit of CR0 and, for some models, adjust the MTRR. Coherency and invalidation concerns apply.

[38] In light of the Cell's high parallelism and the SPE's abundance of 128-bit registers (which can be effectively utilized by bitslice implementations), it seems to have considerable performance potential in cryptographic and cryptanalytic applications.

[39] If the tables stay static for long then the attacker can locate them (see Section 8.3.6) and discern their organization. This was prematurely dismissed by Lauradoux [115], who assumed that the mapping of table entries to memory storage will be attacked only by exhaustive search over all possible such mappings; the mapping can be recovered efficiently on an entry-by-entry basis.

Another variant is to mix the order of the table elements several times during each encryption. The permutations need to be chosen with lookup efficiency in mind (e.g., via a linear congruential sequence), and the choice of permutation needs to be sufficiently strong; in particular, it should employ entropy from an external source (whose availability is application-specific).[40]

The performance and security of this approach are very architecture-dependent. For example, the required strength of the pseudorandom sequence and frequency of randomization depend on the maximal probing frequency feasible for the attacker.

8.5.9 Hiding the timing

All of our attacks perform timing measurements, whether of the encryption itself (in Section 8.3.4) or of accesses to the attacker's own memory (in all other cases). A natural countermeasure for timing attacks is to try to hide the timing information. One common suggestion for mitigating timing attacks is to add noise to the observed timings by adding random delays to measured operations, thereby forcing the attacker to perform and average many measurements. Another approach is to normalize all timings to a fixed value, by adding appropriate delays to the encryption, but beside the practical difficulties in implementing this, it means all encryptions have to be as slow as the worst-case timing (achieved here when all memory accesses miss the cache). Neither of these provide protection against the Prime+Probe synchronous attack or the asynchronous attack.

At the operating system or processor level, one can limit the resolution or accuracy of the clock available to the attacker; as discussed by Hu [86], this is a generic way to reduce the bandwidth of side channels, but is non-trivial to achieve in the presence of auxiliary timing information (e.g., from multiple threads [167]), and will unpredictably affect legitimate programs that rely on precise timing information. The attacker will still be able to obtain the same information as before by averaging over more samples to compensate for the reduced signal-to-noise ratio. Since some of our attacks require only a few milliseconds of measurements, to make them infeasible the clock accuracy may have to be degraded to an extent that interferes with legitimate applications.

8.5.10 Selective round protection

The attacks described in Sections 8.3 and 8.4 detect and analyze memory accesses in the first two rounds (for known input) or last two rounds (for known output). To protect against these specific attacks it suffices to protect those four rounds by some of the means given above (i.e., hiding, normalizing or preventing memory accesses), while using the faster, unprotected implementation for the internal rounds.[41] This does not protect against other cryptanalytic techniques that can be employed using the same measurement methods. For example, with chosen plaintexts, the

[40] Some of these variants were suggested to us by Intel Corp, and implemented in [36], following an early version of this work.

[41] This was suggested to us by Intel Corp, and implemented in [36], following an early version of this work.

table accesses in the 3rd round can be analyzed by differential cryptanalysis (using a 2-round truncated differential). None the less, those cryptanalytic techniques require more data and/or chosen data, and thus when quantitatively balancing resilience against cache-based attacks and performance, it is sensible to provide somewhat weaker protection for internal rounds.

8.5.11 Operating system support

Several of the countermeasures suggested above require privileged operations that are not available to normal user processes in general-purpose operating systems. In some scenarios and platforms, these countermeasures may be superior (in efficiency or safety) to any method that can be achieved by user processes. One way to address this is to provide secure execution of cryptographic primitives as operating system services. For example, the Linux kernel already contains a modular library of cryptographic primitives for internal use; this functionality could be exposed to user processes through an appropriate interface. A major disadvantage of this approach is its lack of flexibility: support for new primitives or modes will require operating system modifications (or loadable drivers) which exceed the scope of normal applications.

An alternative approach is to provide a secure execution facility to user processes.[42] This facility would allow the user to mark a "sensitive section" in his code and ask the operating system to execute it with a guarantee: either the sensitive section is executed under a promise sufficient to allow efficient execution (e.g., disabled task switching and parallelism, or cache in "no-fill" mode — see above), or its execution fails gracefully. When asked to execute a sensitive section, the operating system will attempt to put the machine into the appropriate mode for satisfying the promise, which may require privileged operations; it will then attempt to fully execute the code of the sensitive section under the user's normal permissions. If this cannot be accomplished (e.g., a hardware interrupt may force task switching, normal cache operation may have to be enabled to service some performance-critical need, or the process may have exceeded its time quota) then the execution of the sensitive section will be aborted and prescribed cleanup operations will be performed (e.g., complete cache invalidation before any other process is executed). The failure will be reported to the process (now back in normal execution mode) so it can restart the failed sensitive section later.

The exact semantics of this "sensitive section" mechanism depend on the specific countermeasure and on the operating system's conventions. This approach, while hardly the simplest, offers maximal flexibility to user processes; it may also be applicable inside the kernel when the promise cannot be guaranteed to hold (e.g., if interrupts cannot be disabled).

The impact of these attacks is summarized in Chapter 10.

[42] Special cases of this were discussed in [167] and [23], though the latter calls for this to be implemented at the CPU hardware level.

Chapter 9

Acoustic cryptanalysis

9.1 Introduction

9.1.1 Overview

The human ear is a highly effective device for collecting inadvertent acoustic outputs from adversarial systems — originally, prey and predators, but in modern times of need, such as the 1956 Suez crisis, even a mechanical Hagelin cipher machine may be subject to unanticipated auricular attention [223]. However, it was widely believed that acoustic eavesdropping on electronic computers, whether by ear or by microphone, is not relevant against multi-GHz circuits[1]; at best one could hope to eavesdrop on mechanical I/O devices or on very low-bandwidth channels such as hard disk head movements.

We demonstrate that unintended acoustic signals do, in fact, leak a wealth of information on computation in many modern computers. These signals emanate from components in the power regulation circuitry, which are modulated by the CPU's workload. Most basically, we can easily differentiate between heavy computation and idleness with good temporal resolution; this, by itself, suffices to apply key-recovery timing attacks via an acoustic feedback channel. By further analysis of acoustic spectral signatures, we demonstrate the ability to identify internal details of secret-key RSA operations (e.g., operations modulo the different secret primes), and to easily distinguish between different secret keys based on their acoustic spectral signatures.

9.1.2 Related works

Auditory eavesdropping on human conversations is a common practice, whose first publications date back several millenia [78]. Analysis of sound emanations from mechanical devices is a newer

[1] Sound propagation in air has a useful range in frequencies of at most a few hundred KHz, due to non-linear attenuation and distortion effects, such as viscosity, relaxation and diffusion at the molecular level. The exact attenuation rates depend on frequency, air pressure, temperature and humidity; see e.g. [60][21].

affair, bearing precedents in military context such as identification of watercrafts via the sound signature of their engine and propeller as recorded by hydrophones. There are anecdotal stories of computer programmers and system administrators monitoring the high-level behavior of their systems by listening to sound cues generated by mechanical system components, such as hard disk head seeks; but these do not appear very useful in the presence of caching, delayed writes and multitasking. Another such leakage source lies in coupling of system components to internal audio circuitry via unintended electric or electromagnetic radiation channels; but well-built systems keep this cross-talk at a negligible level.

In the context of information security, Wright [223, pp. 103–107] provides the aforementioned account of MI5 and GCHQ using a phone tap to eavesdrop on a Hagelin cipher machine in an Egyptian embassy, thereby counting the number of clicks during the rotors' secret setting procedure.[2] The sound of keystrokes on keyboards has been observed to leak information about the pressed keys, as well as the identity of the user, due to timing patterns. Recent research (concurrent to ours) by Asonov and Agrawal [13], improved by Zhuang [224], shows that keys can also be distinguished individually by their sound, due to minute differences in mechanical properties such as keys' position on a slightly vibrating printed circuit board. While these attacks are applicable to data that is entered manually (e.g., passwords), they are not applicable to larger secret data such as RSA keys.

9.2 Results

9.2.1 Experimental setup

We used a variety of measurement configurations and subjects. Where not stated otherwise, the recordings mentioned in this chapter were conducted as follows. To demonstrate the feasibility of the attack, we employed inexpensive off-the-shelf equipment: a Røde NT3 condenser microphone (US$170), an Alto S-6 mixer (US$55) serving as an amplifier and rudimentary equalizer, and a Creative Labs Audigy 2 sound card (US$70) for recording into a measurement computer. The recordings were made in normal office conditions, with the microphone placed 20cm from the recorded subject computer. A weak high-pass prefilter (roughly -10dB below 1KHz, +10dB above 10KHz) was applied using the mixer's rudimentary built-in equalizer. The subject of the recording is an unbranded desktop computer using a PC Chips M754LMR motherboard, an Intel Celeron 666MHz CPU and an Astec ATX200-3516 power supply. The PC case was opened and fan noise was quenched by disconnecting the fans. This computer was chosen for its particularly clear acoustic emanations, but is by no means a special case: every computer we tested showed significant correlation between acoustic spectrum and CPU activities, and in about half the cases the effect could be heard even by naked ear when using appropriate CPU activity patterns.[3]

[2] Wright refers to this technique as bearing the codename "ENGULF" [223, p. 107].

[3] Anecdotal feedback we have received following the initial publication of this research confirms the ubiquity of the effect.

Comparable results were achieved under more realistic conditions, where the subject computer is fully assembled and placed 1m–2m from the microphone, using studio-grade audio equipment. For example, using a high-quality analog high-pass filter to attenuate strong low-frequency noise (e.g., fan humming) allows further amplification of interesting signals before analog-to-digital quantization.

We also acquired advanced lab-grade recording equipment, capable of higher frequencies and sensitivity: a Brüel&Kjær type 4939 1/4" microphone with a type 2670 amplifier and a NEXUS conditioning amplifier, and a National Instruments PCI-6052E DAQ PCI card (total value US$6,500).[4] The Brüel&Kjær microphone is rated for uniform sensitivity up to 100KHz, and offers fair sensitivity up to roughly 160KHz, matching the Nyquist frequency of the 333KHz sampling rate provided by the DAQ card.[5].

The spectrograms below depict the empirical measurements, processed and rendered via a sliding-window FFT using the Baudline [195] signal analysis software. To make the relevant signals stand out, we applied digital equalization (a low-pass filter to attenuate low frequencies) and a post-FFT intensity histogram adjustment.

9.2.2 The sound of RSA signatures

We begin by inspecting the sound generated during RSA secret-key operations, i.e., signing and decryption. We chose a common RSA implementation, namely GnuPG 1.2.4 [67], and used it to sign short messages using randomly-generated 4096-bit RSA keys. Figure 9.1 depicts the spectrogram of two identical signing operations in sequence, using the same key and message. Each signing operation is preceded by a short delay during which the CPU is in a sleep state.

Several effects are evident. The delays, where the computer is idle, are manifested as bright horizontal strips, i.e., wideband noise.[6] Between these strips, the two signing operations are clearly distinguished. Halfway through each signing operation there is a transition at several frequency bands. This corresponds to a detail in the RSA implementation of GnuPG: for public key $n = pq$, the RSA signature $s = m^d \pmod n$ is computed by evaluating $m^{d \bmod (p-1)} \pmod p$ and $m^{d \bmod (q-1)} \pmod q$, and combining these via the Chinese Remainder Theorem. The first half of the signing operation corresponds to the exponentiation modulo p, and the second to the exponentiation modulo q.[7] Strikingly, the transition between these secret exponents is clearly visible. This effect is consistent and reproducible; indeed, the two acoustic signatures depicted

[4] The DAQ card was graciously donated for this research by National Instruments Israel.

[5] We were able to record even higher frequencies using this equipment, by foregoing the use of low-pass filters and exploiting the aliasing effects in the DAQ.

[6] The delay is implemented by the operating system using the x86 `HLT` instruction, which puts the CPU into a special low-power sleep state that lasts until the next hardware interrupt. On modern CPUs this temporarily shuts down many of the on-chip circuits, which dramatically lowers power consumption and alters acoustic emissions for relatively long time.

[7] This was confirmed by patching the GnuPG code to just repeat the operation modulo p twice, which indeed eliminated the effect from the measurements.

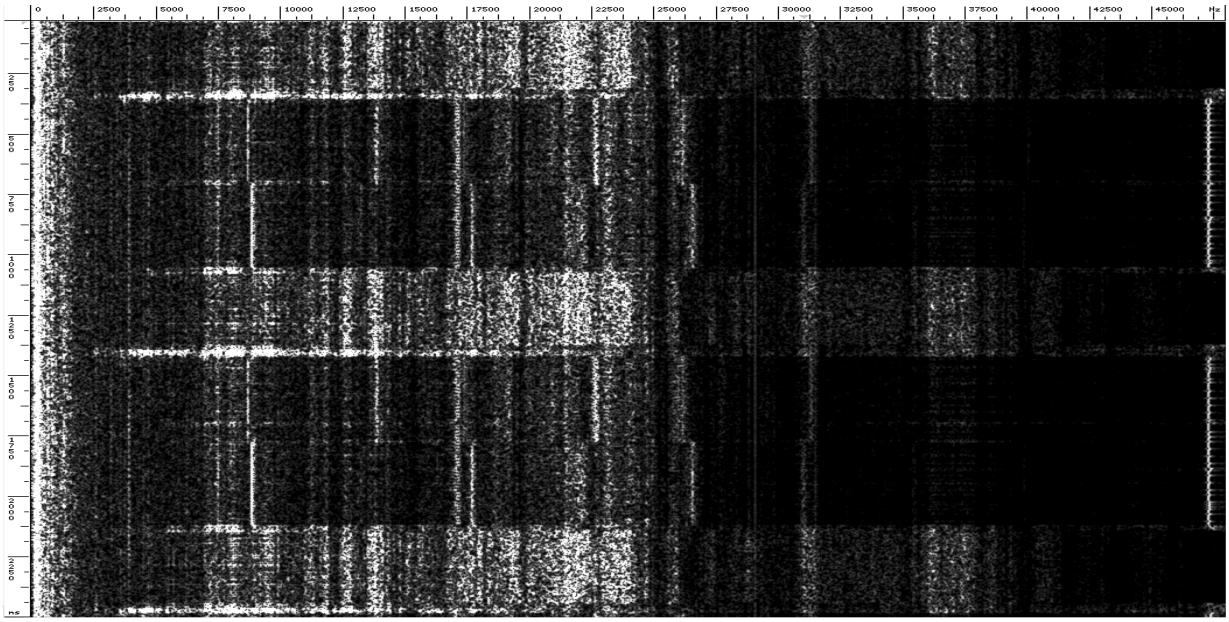

Figure 9.1: Spectrogram of an acoustic measurement of two 4096-bit GnuPG RSA signatures. The horizontal axis is frequency (0-48KHz), the vertical axis is time (0-2.5sec), and intensity is proportional to the instantaneous energy in that frequency band.

in Figure 9.1 are very similar. RSA decryption operations use an essentially identical algorithm, with similar results.

9.2.3 Distinguishing between RSA secret keys

Beside measuring the duration and internal properties of individual RSA secret-key operations, we have investigated the effect of different keys. Having observed that the acoustic signature of modular integer exponentiation depends on the modulus involved, one may expect different keys to cause different sounds. This is indeed the case, as demonstrated in Figure 9.2. Here, we used GnuPG 1.2.4 to sign a fixed message using 7 different 4096-bit RSA keys randomly generated beforehand. Each signature is preceded by a short CPU sleep. It is readily observed that each signature (and in fact, each exponentiation using modulus p or q) has a unique spectral signature.

This ability to distinguish keys is of interest in traffic-analysis applications.[8] It is likewise possible to distinguish between algorithms, between different implementations of an algorithm and and between different computers (even of the same model) running the same algorithm. Furthermore, the leaked key information may be exploitable for a systematic key recovery attack.[9]

[8]For example, observing that an embassy has now decrypted a message using a rarely employed key, heard before only in specific diplomatic circumstances, can form valuable information.

[9]We are presently pursuing this extension.

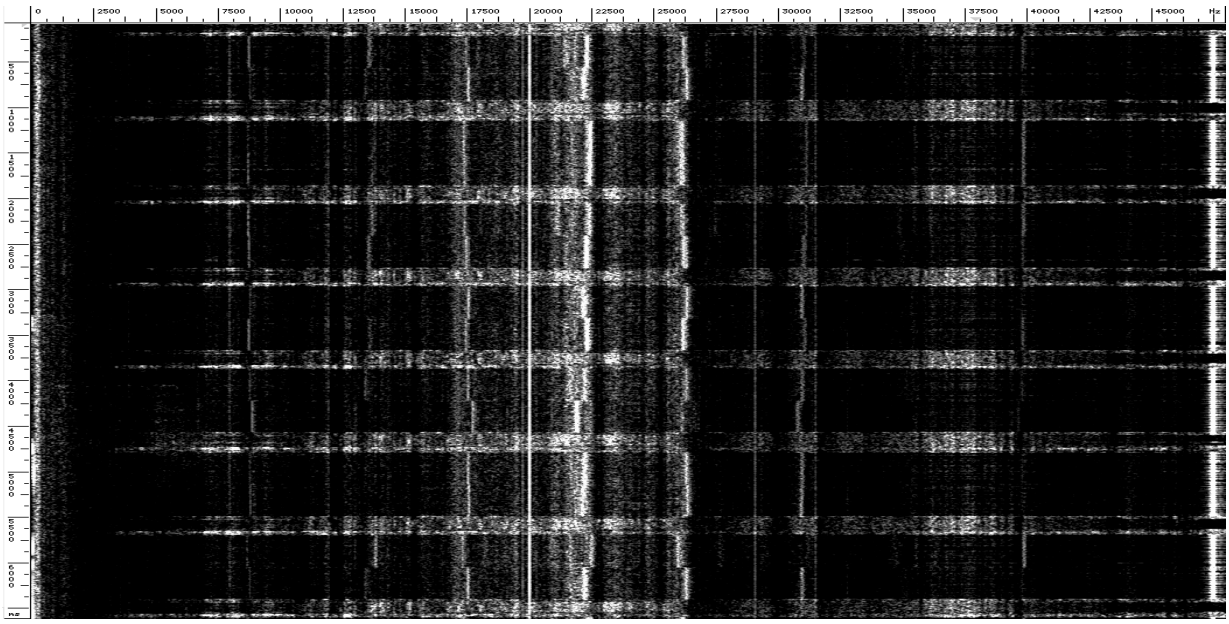

Figure 9.2: Spectrogram of an acoustic measurement of different GnuPG RSA signatures using 7 different keys.

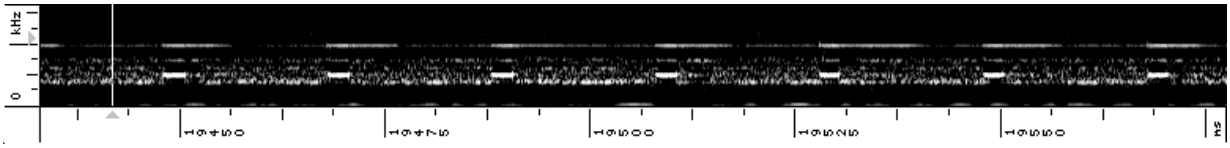

Figure 9.3: Spectrogram of an acoustic measurement of brief bursts of computation (3ms each, with 20ms period) on an otherwise idle CPU, using the lab-grade equipment. Unlike the rest of the figures, here time is horizontal and frequency is vertical.

9.2.4 Timing attacks

We were able to (manually but reliably) distinguish a cryptographic operation from idleness with a temporal resolution close to 1ms using the off-the-shelf equipment. Using the lab-grade equipment, the temporal resolution was under 0.4ms (see Figure 9.3). This resolution suffices for carrying key-recovery timing attacks (see §7.3) via an acoustic feedback channel. It thus facilitates timing attacks in situations where a network feedback channel is not available for timing measurements, e.g., in the case of one-way transmission of e-mail or when one-way communication is enforced by physical protocol such as unidirectional transfer of media. Furthermore, the ability to measure the duration of operations modulo p vs. modulo q separately (see §9.2.2) allows an improvement in the efficiency of the timing attacks.

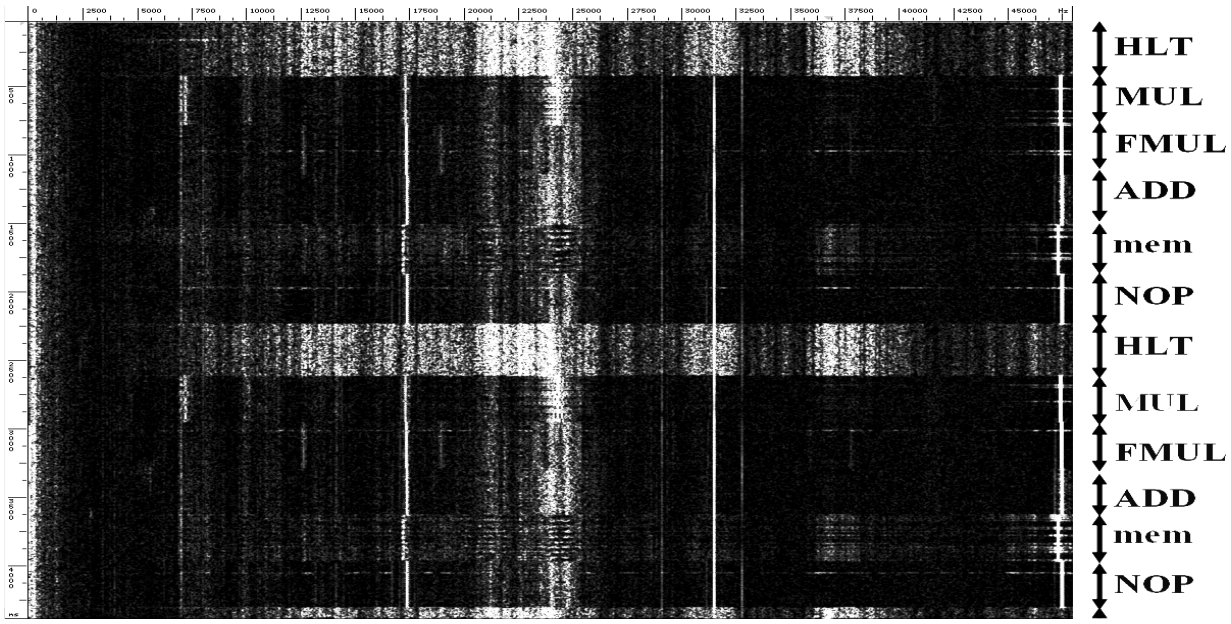

Figure 9.4: Acoustic measurement of different CPU instructions

9.2.5 Instruction pattern differentiation

Turning to a lower-level investigation of the effect, we observe a difference between characteristic spectra of different CPU operations. To demonstrate this, we wrote a simple program that executes (partially unrolled) loops containing one of the following x86 instructions: `HLT` (CPU sleep), `MUL` (integer multiplication), `FMUL` (floating-point multiplication), main memory access (forcing L1 and L2 caches misses), and `REP NOP` (short-term idle). Figure 9.4 shows a recording of a series of such homogeneous loops.

Each of the instruction loops can be distinguished from the others by some characteristic of its spectrum.[10] The differences are even more pronounced at the higher frequencies accessible via the lab-grade equipment. Heterogeneous operations of instructions likewise have a characteristic sound signature, and can often be distinguished from each other. We conclude that the acoustic leakage divulges information about what program is running and what routines are frequently executed.

[10]The `ADD` and `REP NOP` instructions are in this case distinguished only by a subtle difference in the intensity of the 24–25KHz band, possibly because pure integer addition exercises a tiny fraction of the CPU circuit.

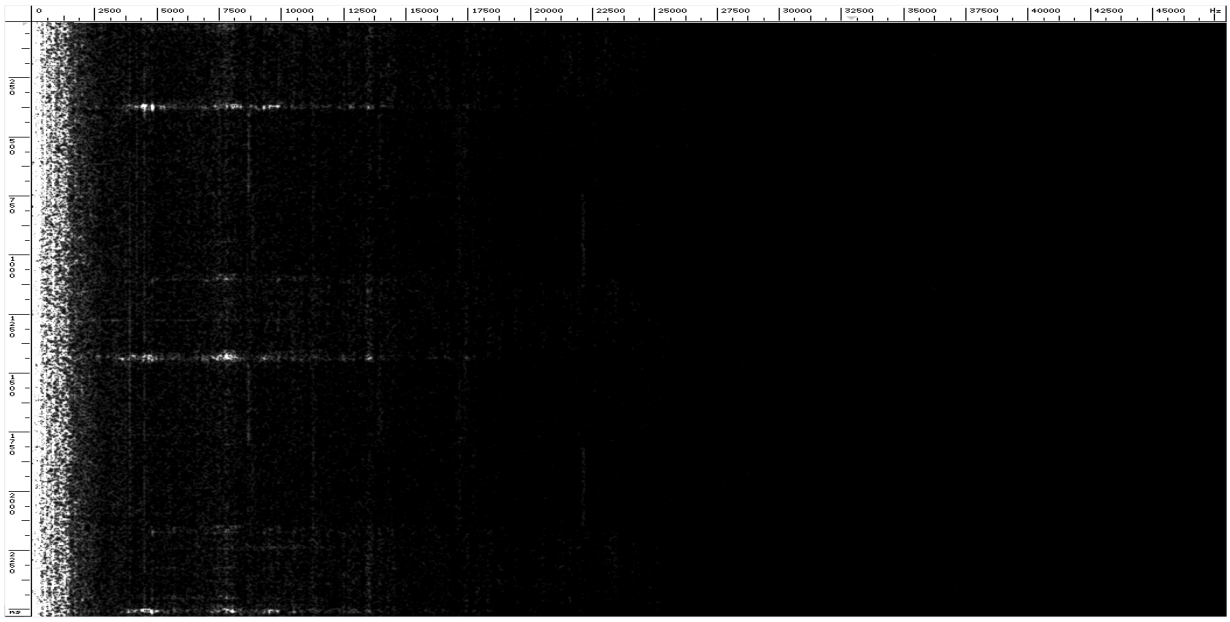

Figure 9.5: Spectrogram of an acoustic measurement of two 4096-bit GnuPG RSA signatures, with the microphone blocked by cloth. The signal is nearly eliminated.

9.2.6 Verifying acoustic transduction

A potential concern in our experiments is the possibility that the recorded signal is not transmitted by sound waves in air, but rather by electromagnetic radiation picked up by some part of the recording equipment and further amplified (e.g., the microphone assembly or cable serving as antennae despite ample grounding). This would have relegated the attack to the well-investigated realm of electromagnetic side-channels.

The fact that some of the effects are directly audible by a human listener indicates that some acoustic transduction is indeed taking place, but does not ensure its magnitude and spectral properties. To test the hypothesis conclusively, we repeated the experiment depicted Figure 9.1, except that the microphone was muffled via a device that is absorbant acoustically but nearly transparent electromagnetically.[11] The corresponding recording is given in Figure 9.5. As expected, all but the lowest frequencies are greatly attenuated, and the signal of interest is gone. This confirms that the signal is indeed transmitted by acoustic transduction. Furthermore, if the microphone is turned off (using its built-in switch) but left connected to a running amplifier, the amplifier's output is at its noise floor.

[11] Namely, a folded dry cloth handkerchief. Woven synthetic fiber yielded the same effect.

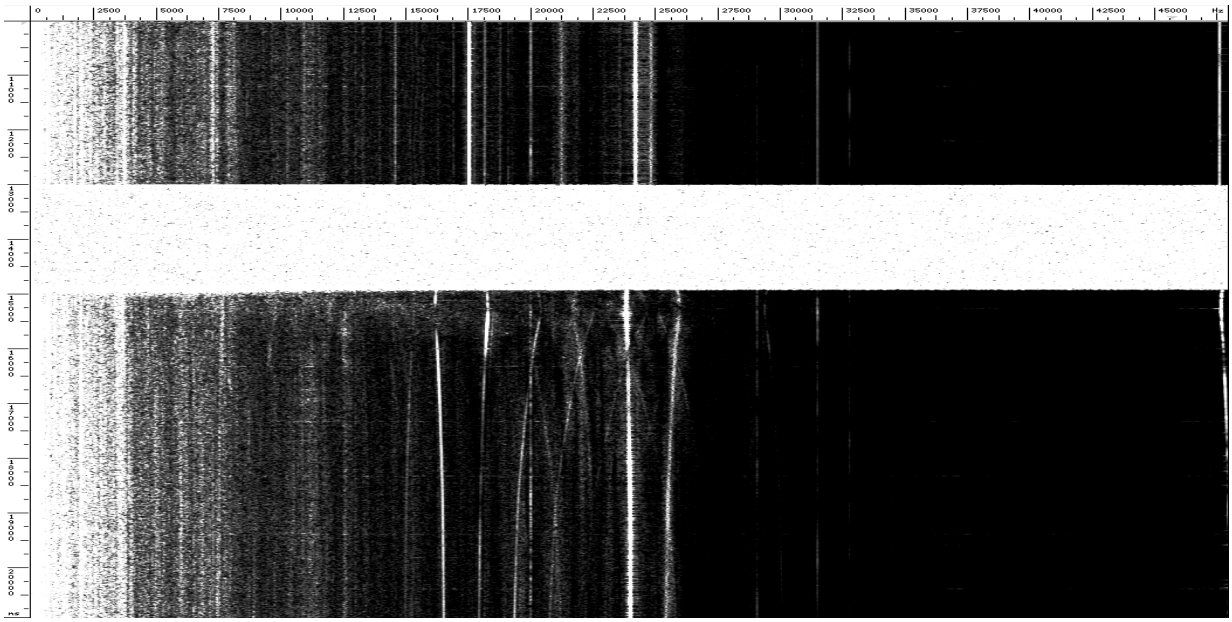

Figure 9.6: Acoustic measurement of a MUL loop during cooling of power-supply capacitors on PC Chips M754LMR motherboard.

9.2.7 Source of acoustic emanations

To identify the source of the acoustic emanations, we partially disassembled a subject computer and perturbed its various components one by one (by local application of Quik-Freeze spray), all the while looking for a significant effect on the recorded sound. The culprit identified by this method is a bank of 1500μF capacitors near the CPU socket of the PC Chips M754LMR motherboard. Figure 9.6 depicts the effect of cooling these capacitors while the CPU is executing a loop of MUL instructions. Following the spray application (which saturates the recording), the information-bearing spectral lines are significantly shifted. No such effect was observed for any other component.

We surmise that the sound is generated in the power regulation circuitry — either the capacitors themselves, or the associated and adjacent inductor coils (both capacitors and inductors are known to be prone to vibrations). This circuitry is modulated by the CPU's momentary load, and a modern CPU can vary its load by an order of magnitude in a few microseconds. While vibration is most common in capacitors whose electrolyte has dried out, our experiments show that information-bearing emanations are often generated by brand-new desktop and laptop computers.

9.3 Countermeasures

One obvious countermeasure is to use sound dampening equipment (i..e., "sound-proof" boxes or rooms) that is designed to sufficiently attenuate all relevant frequencies. Conversely, a sufficiently strong wide-band noise source can mask the informative signals (but ergonomic concerns may render this unattractive). Careful circuit design and high-quality electronic components can reduce the emanations. Specifically for certain cryptographic algorithms and at some cost in performance, one can employ algorithmic techniques to reduce the cryptanalytic usefulness of the emanations to attacker. A variety of masking and normalization techniques have been designed to protect cryptographic code against power and electromagnetic side-channel leakages; these would be effective also against the acoustic side channel, whose bandwidth is significantly lower. However, these techniques often have a nontrivial cost in performance and complexity.

Notably, the acoustic attack would not be foiled merely by the presence of typical fan and room noises. The interesting acoustic signals are mostly above 10KHz, whereas typical computer fan noise and normal room noise are concentrated at lower frequencies and can thus be filtered out by suitable equipment. Multitasking does not pose a large challenge either, since different tasks can be distinguished by their different acoustic spectral signatures. The presence of multiple computers does not significantly hamper the attack either, since they can be told apart by their different acoustic signatures, which vary with the hardware, software, component temperatures and environmental conditions.

Chapter 10

Conclusions and implications of Part II

10.1 Summary of results

In Chapter 8, we described novel attacks which exploits inter-process information leakage through the state of the CPU's memory cache. This leakage reveals memory access patterns, which can be used for cryptanalysis of cryptographic primitives that employ data-dependent table lookups. Exploiting this leakage allows an unprivileged process to attack other processes running in parallel on the same processor, despite partitioning methods such as memory protection, sandboxing and virtualization. Some of our methods require only the ability to trigger services that perform encryption or MAC using the unknown key, such as encrypted disk partitions or secure network links. Moreover, we demonstrated an extremely strong type of attack, which requires knowledge of neither the specific plaintexts nor ciphertexts, and works by merely monitoring the effect of the cryptographic process on the cache. We discussed in detail several such attacks on AES, and experimentally demonstrated their applicability to real systems, such as OpenSSL and Linux's `dm-crypt` encrypted partitions (in the latter case, the full key was recovered after just 800 writes to the partition, taking 65 milliseconds). Finally, we proposed a variety of countermeasures.

In Chapter 9 we demonstrated attacks that exploit acoustic emanations from modern computers, wherein the power circuitry creates vibrations that are modulated by CPU activity. We demonstrated acoustic leakage of secret information from sensitive computation such as RSA signing and decryption. These leakages are sufficient for distinguishing attacks on RSA, and provide strong evidence that key recovery may be possible.

10.2 Vulnerable cryptographic primitives

10.2.1 Cache attacks

The cache attacks we have demonstrated are particularly effective for typical implementations of AES, for two reasons. First, the memory access patterns have a simple relation to the inputs; for example, the indices accessed in the first round are simply the XOR of a key byte and a plaintext byte. Second, the parameters of the lookup tables are favorable: there is a large number of memory blocks involved (but not too many to exceed the cache size) and thus many bits are leaked by each access. Moreover, there is a significant probability that a given memory block will not be accessed at all during a given random encryption.

Beyond AES, such attacks are potentially applicable to any implementation of a cryptographic primitive that performs key- and input-dependent memory accesses. The efficiency of the attack depends heavily on the structure of the cipher and chosen implementation, but heuristically, large lookup tables increase the effectiveness of all attacks: having few accesses to each table helps the synchronous attacks, whereas the related property of having temporally infrequent accesses to each table helps the asynchronous attack. Large individual table entries also aid the attacker, in reducing the uncertainty about which table entry was addressed in a given memory block. This is somewhat counterintuitive, since it is usually believed that large S-boxes are more secure.

For example, DES is vulnerable when implemented using large lookup tables which incorporate the P permutation and/or to compute two S-boxes simultaneously. Cryptosystems based on large-integer modular arithmetic, such as RSA, can be vulnerable when exponentiation is performed using a precomputed table of small powers (see [167]). Moreover, a naive square-and-multiply implementation would leak information through accesses to long-integer operands in memory. The same potentially applies to ECC-based cryptosystems.

Primitives that are normally implemented without lookup tables, such as the SHA family [150] and bitsliced Serpent [10], are impervious to the attacks described here. However, to protect against timing attacks one should scrutinize implementations for use of instructions whose timing is key- and input-dependent (e.g., bit shifts and multiplications on some platforms) and for data-dependent execution branches (which may be analyzed through data cache access, instruction/trace cache access or timing). Note that timing variability of non-memory operations can be measured by an unrelated process running on the same machine, using a variant of the asynchronous attack, via the effect of those operations on the scheduling of memory accesses.

10.2.2 Acoustic attacks

Acoustic attacks appear effective in the case of code consiting of long, relatively uniform patterns of instructions, as is typical of public-key cryptography with large keys. Due to the limited temporal resolution of this channel, we do not expect much useful information from brief operations such as a single invocation of a typical block cipher (although repeated operation may

yield key-dependent activation patterns, e.g., due to cache contention). An exact characteristic is difficult since the modulation path leading to acoustic emanations is imperfectly understood and hardware-dependent.

10.2.3 Non-cryptographic systems

The cache and acoustic leakages we have identified apply to any operation, cryptographic or otherwise. Above we have focused on cryptographic operations because these are designed and trusted to protect information, and thus information leakgage from within them can be critical (for example, recovering a single decryption key can compromise the secrecy of all messages sent over the corresponding communication channel). However, information leakage can be harmful also in non-cryptographic context. For example, even knowledge of what programs are running on someone's computer at a given time can be sensitive.

10.3 Attack scenarios

The cache attacks require no dedicated hardware at all, and work in pure software. The acoustic attacks are also quite accessible: we have demonstrated that easily obtained sound recording equipment, at a cost of a few hundred dollars, will in many cases suffice to pick up a wealth of information from useful range. We believe that even microphones built into mobile phones and laptops would often suffice.

These novel attacks are easy to deploy using pure software or minimal hardware, and affect a large number of deployed systems — including those that are otherwise hardened or insusceptible to previously known attacks. This, in effect, demonstrates that side-channel attacks have been "commoditized" — they no longer lie solely in the realm of government agencies and professional espionage, but have now become an important design consideration for security even in home and office environment.

At the system level, cache state analysis is of concern in essentially any case where process separation is employed in the presence of malicious code. This class of systems includes many multi-user systems, as well as web browsing, DRM applications, the Trusted Computing Platform [204] and NGSCB [140]. The same applies to acoustic cryptanalysis, whenever malicious code can access a nearby microphone device and thus record the acoustic effects of other local processes.

Disturbingly, virtual machines and sandboxes offer little protection against the asynchronous cache attack (in which attacker needs only the ability to access his own memory and measure time) and against the acoustic attacks (if the attacker gains access to a nearby microphone). Thus, our attacks may cross the boundaries supposedly enforced by FreeBSD `jail()`, VMware [214][1], Xen [208], the Java Virtual Machine [133] and plausibly even scripting language interpreters.

[1] This compromises the system described in a recent NSA patent 6,922,774.[139]

Today's hardware-assisted virtualization technologies, such as Intel's "Virtualization Technology" and AMD's "Secure Virtual Machine, offer no protection either.

Remote cache attacks are in principle possible, and if proven efficient they could pose serious threats to secure network connections such as IPsec [102] and OpenVPN [157].

A notable property of the acoustic side channel is that, uniquely among the physical side channels, the interception equipment is ubiquitous. Unlike radio receivers or photodetectors, microphones are present at nearly every facility, and are built into common equipment such as mobile and landline phones, laptop computers and PDAa. This opens new venues of attack. For example, an unclassified laptop or a PDA phone may be compromised (e.g., via a web browser bug) and programmed to wake up at a given hour and record using its internal microphone. If this device is later carried into a secure computer room (but not connected to anything), it may record valuable acoustic information without its owner's knowledge. Side-channel countermeasures presently deployed in secure facilities may offer little resistance; for example, Faraday cages for computers typically contain air vents covered by grounded meshes, which attenuate RF electromagnetic radiation but are acoustically transparent. The wide repertoire of techniques for eavesdropping on sound, consisting of such powerful methods as laser interferometer microphones capable of detecting sound-induced vibrations in window panes (or other reflective surfaces) from a distance, enables a multitude of additional attack scenarios.

10.4 Mitigation

Cache attacks. We have described a variety of countermeasures against cache state analysis attacks; some of these are generic, while others are specific to AES. However, none of these unconditionally mitigates the attacks while offering performance close to current implementations. Thus, finding an efficient and secure solution that is application- and architecture-independent remains an open problem. In evaluating countermeasures, one should pay particular attention to the asynchronous attacks, which on some platforms allow the attacker to obtain (a fair approximation of) the full transcript of memory accesses done by the cryptographic code.

Acoustic attacks. The acoustic channel is similar to other physical side channels, in that mitigation can be achieved through algorithmic techniques (for specific cryptographic primitives), by a careful circuit design and use of high-quality components, or by thorough shielding and physical access controls.

10.5 Impact and follow-up works

Our work on cache state analysis has spurred significant academic and industry interest. Among the subsequent works are the following:

Countermeasures. Following a pre-publication of this research, Brickell et al. of Intel Corp. [36][37] implemented and experimentally evaluated several AES implementations that reduce the cache side-channel leakage (see discussion in §8.5), and Page [163] evaluated partitioned cache architectures as a countermeasure.

Survey and extensions to related attacks. In [41], Canteaut et al. surveys and classifies the various cache attacks, and proposes some countermeasures and extensions.

Exploiting the OS scheduler. In [153]), Neve and Seifert empirically demonstrate the effectiveness of an extension we have merely alluded to hypothetically: carrying out an asynchronous attacks without simultaneous multithreading, by exploiting only the OS scheduling and interrupts. Indeed, they show that with appropriate setup the result provides excellent temporal resolution. They also demonstrate the effectiveness of analyzing the last round of AES instead of the first one (where applicable).

Collision-based attacks. In [33], Bonneau and Mironov present an attack on AES based on exploiting internal cache collisions, following the approach of Tsunoo et al. (see §8.1.2), and relate it to our attacks. Comparable results were obtained by Acıiçmez et al. [7].

Branch prediction and instruction cache attacks. In [5, 6, 2], Acıiçmez et al. describe new classes of attacks that exploit the CPU *instruction* cache or its branch prediction mechanism, instead of the *data* cache considered herein. They demonstrate efficient RSA key recovery via contention for these resources. The measurement approaches (and hence attack scenarios) are similar to the data cache attack techniques described here, but the information obtained is about the execution path rather than data accesses. Veith et al. [210] presented a related attack, which monitors branch prediction via the CPU performance counters. Since the type of vulnerable code is different compared to data cache attacks, these attacks are complementary.

Publications and statement of originality

Most of the results presented in this dissertation have been published as follows (the papers marked by ⦿ were chosen as the corresponding conference's opening presentation):

Refereed publications:

- ⦿ Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, Eran Tromer, *Analysis of Bernstein's factorization circuit*, proc. Asiacrypt 2002, LNCS 2501, 1–26, Springer, 2002 [131]
- ⦿ Adi Shamir, Eran Tromer, *Factoring large numbers with the TWIRL device*, proc. CRYPTO 2003, LNCS 2729, 1–26, Springer, 2003 [187]
- Arjen K. Lenstra, Eran Tromer, Adi Shamir, Wil Kortsmit, Bruce Dodson, James Hughes, Paul Leyland, *Factoring estimates for a 1024-bit RSA modulus*, proc. Asiacrypt 2003, Springer, LNCS 2894, 331–346, Springer, 2003 [132]
- Willi Geiselmann, Hubert Köpfer, Rainer Steinwandt, Eran Tromer, *Improved routing-based linear algebra for the Number Field Sieve*, proc. International Conference on Information Technology: Coding and Computing (ITCC'05), vol. 1, 636-641, IEEE, 2005 [70]
- Willi Geiselmann, Adi Shamir, Rainer Steinwandt, Eran Tromer, *Scalable hardware for sparse systems of linear equations, with applications to integer factorization*, proc. CHES 2005, LNCS 3659, 131–146, Springer, 2005 [72]
- ⦿ Adi Shamir, Dag Arne Osvik, Eran Tromer, *Cache attacks and countermeasures: the case of AES*, proc. RSA Conference Cryptographers Track (CT-RSA) 2006, LNCS 3860, 1–20, Springer, 2006; first presented at the FSE'05 rump session, February 2005 [186]

Invited publications:

- Adi Shamir, Eran Tromer, *On the cost of factoring RSA-1024*, RSA CryptoBytes, vol. 6 no. 2, 10–19, 2003 [188]
- ⦿ Adi Shamir, Eran Tromer, *Special-purpose hardware for factoring: the NFS sieving step*, proc. Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS), 2005 [190]

- Willi Geiselmann, Adi Shamir, Rainer Steinwandt, Eran Tromer, *Scalable hardware for sparse systems of linear equations, with applications to integer factorization*, proc. CHES 2005, LNCS 3659, 131–146, Springer, 2005 [71]
- Willi Geiselmann, Adi Shamir, Rainer Steinwandt, Eran Tromer, *Fault-tolerance in hardware for sparse systems of linear equations, with applications to integer factorization*, Chapter 8 in N. Nedjah, L. de Macedo Mourelle (Eds.), New Trends in Cryptographic Systems, Nova Science Publishers, 2006 [73]

The results in Chapter 9 are joint work (in progress) with Adi Shamir, announced at the Eurocrypt'04 rump session [189][191].

The dissertation contains additional results beyond the aforementioned publications.

All results presented in this dissertation are, where not stated otherwise, original research. Most of this research was done in collaboration with other investigators, as reflected in the aforementioned publications. My personal contribution to each of these publications was substantial at the conceptual, technical and editorial levels.

Bibliography

[1] Martin Abadi, Mike Burrows, Mark Manasse, Ted Wobber, *Moderately hard, memory-bound functions*, ACM Transactions on Internet Technology, vol. 5, issue 2, 299–327, 2005

[2] Onur Acıiçmez, *Yet another microarchitectural attack: exploiting I-cache*, IACR Cryptology ePrint Archive, report 2007/164, 2007, `http://eprint.iacr.org/2007/164`

[3] Onur Acıiçmez, Çetin Kaya Koç, *Trace driven cache attack on AES*, IACR Cryptology ePrint Archive, report 2006/138, 2006, `http://eprint.iacr.org/2006/138`; full version of [4]

[4] Onur Acıiçmez, Çetin Kaya Koç, *Trace driven cache attack on AES (short paper)*, proc. International Conference on Information and Communications Security (ICICS) 2006, LNCS 4296, 112–121, Springer, 2006; short version of [3]

[5] Onur Acıiçmez, Çetin Kaya Koç, Jean-Pierre Seifert, *On the power of simple branch prediction analysis*, IACR Cryptology ePrint Archive, report 2006/351, 2006

[6] Onur Acıiçmez, Çetin Kaya Koç, Jean-Pierre Seifert, *Predicting secret keys via branch prediction*, proc. RSA Conference Cryptographers Track (CT-RSA) 2007, LNCS 4377, 225–242, Springer, 2007

[7] Onur Acıiçmez, Werner Schindler, Çetin Kaya Koç, *Cache based remote timing attack on the AES*, proc. RSA Conference Cryptographers Track (CT-RSA) 2007, to appear.

[8] Manindra Agrawal, Neeraj Kayal, Nitin Saxena, *PRIMES is in P*. Annals of Mathematics 160, vol. 2, 781–793, 2004

[9] Miklós Ajtai, Cynthia Dwork, *A public-key cryptosystem with worst-case/average-case equivalence*, proc. 29th Annual IEEE Symp. on Foundations of Computer Science (FOCS), 284-293, 1997

[10] Ross J. Anderson, Eli Biham, Lars R. Knudsen, *Serpent: A proposal for the Advanced Encryption Standard*, AES submission, 1998, `http://www.cl.cam.ac.uk/~rja14/serpent.html`

[11] Ross J. Anderson, Markus G. Kuhn, *Soft tempest — an opportunity for NATO*, proc. Information Systems Technology (IST) Symposium "Protecting NATO Information Systems in the 21st Century", Washington DC, 25–27, 1999

[12] Kazumaro Aoki, Yuji Kida, Takeshi Shimoyama, Hiroki Ueda, *GNFS Factoring Statistics of RSA-100, 110, ..., 150*, IACR Cryptology ePrint Archive, report 2004/095, 2006, `http://eprint.iacr.org/2004/095`

[13] Dmitry Asonov, Rakesh Agrawal, *Keyboard acoustic emanations*, proc. IEEE Symposium on Security and Privacy, 3–11, IEEE, 2004

[14] E. Bach, R. Peralta, *Asymptotic semi-smoothness probabilities*, University of Wisconsin, Technical report #1115, October 1992

[15] Friedrich Bahr, M. Böhm, Jens Franke, Thorsten Kleinjung, *rsa200*, e-mail announcement, May 2005, `http://www.loria.fr/~zimmerma/records/rsa200`

[16] Friedrich Bahr, M. Böhm, Jens Franke, Thorsten Kleinjung, *RSA640*, e-mail announcement, November 2005, `http://www.loria.fr/~zimmerma/records/rsa640`

[17] Friedrich Bahr, Jens Franke, Thorsten Kleinjung, M. Lochter, M. Böhm, *RSA-160*, e-mail announcement, Apr. 2003, `http://www.loria.fr/~zimmerma/records/rsa160`

[18] Sashisu Bajracharya, Deapesh Misra, Kris Gaj, Tarek El-Ghazawi, *Reconfigurable hardware implementation of mesh routing in Number Field Sieve factorization*, proc. Field Programmable Technology (FPT) 2004, 263–270, 2004; updated in [19].

[19] Sashisu Bajracharya, Deapesh Misra, Kris Gaj, Tarek El-Ghazawi, *Reconfigurable hardware implementation of mesh routing in Number Field Sieve factorization*, Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'05), 2005; updated version of [18].

[20] E. Barbin, J. Borowczyk, J.-L. Chabert, M. Guillemot, A. Michel-Pajus, A. Djebbar, J.-C. Martzloff, Jean-Luc Chabert, C. Weeks, *A History of Algorithms: From the Pebble to the Microchip*, Springer, 1999

[21] H. E. Bass, Roy G. Keeton, *Ultrasonic absorption in air at elevated temperatures*, Journal of the Acoustical Society of America, vol. 58, 110-112, 1975

[22] Daniel J. Bernstein, *Circuits for integer factorization: a proposal*, manuscript, 2001, `http://cr.yp.to/papers.html`

[23] Daniel J. Bernstein, *Cache-timing attacks on AES*, preprint, 2005, `http://cr.yp.to/papers.html#cachetiming`

[24] Daniel J. Bernstein, Arjen K. Lenstra, *A general number field sieve implementation*, 103–126 in [129], 1993

[25] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, Gianluca Palermo, *AES power attack based on induced cache miss and countermeasure*, proc. International Conference on Information Technology: Coding and Computing (ITCC'05), 586–591, IEEE, 2005

[26] Eli Biham, *A fast new DES implementation in software*, proc. FSE 1997, LNCS 1267, 260–272, Springer, 1997

[27] Eli Biham, Adi Shamir, *Differential fault analysis of secret key cryptosystems*, proc. CRYPTO'97, LNCS 1294, 513– 525, Springer, 1977

[28] Eli Biham, Adi Shamir, *Differential cryptanalysis of DES-like Cryptosystems*, Journal of Cryptology, vol. 4, vo. 1, 3–72, 1991

[29] Lenore Blum, Manuel Blum, Michael Shub, *Comparison of two pseudo-random number generators*, proc. CRYPTO '82, 61-78, Plenum Press, 1983

[30] Manuel Blum, William Evans, Peter Gemmell, Sampath Kannan, Moni Noar, *Checking the correctness of memories*, proc. Conference on Foundations of Computer Science (FOCS) 1991, 90–99, IEEE, 1991

[31] Dan Boneh, David Brumley, *Remote timing attacks are practical*, proc. 12th USENIX Security Symposium, 2003

[32] Dan Boneh, Richard A. DeMillo, Richard J. Lipton, *On the importance of checking cryptographic protocols for faults*, Journal of Cryptology, Springer, vol. 14, no. 2, 101–119, 2001

[33] Joseph Bonneau, Ilya Mironov, *Cache-collision timing attacks against AES*, proc. Cryptographic Hardware and Embedded Systems (CHES) 2006, 201–215, 2006

[34] Richard P. Brent, *Parallel algorithms in linear algebra*, proc. Second NEC Research Symposium (1991), SIAM, 1993, `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub128.html`

[35] Richard P. Brent, *Recent progress and prospects for integer factorisation algorithms*, proc. Proc. Sixth Annual International Computing and Combinatorics Conference, LNCS 1858, 3–22, Springer, 2000

[36] Ernie Brickell, Gary Graunke, Michael Neve, Jean-Pierre Seifert, *Software mitigations to hedge AES against cache-based software side channel vulnerabilities*, IACR Cryptology ePrint Archive, report 2006/052, 2006, `http://eprint.iacr.org/2006/052`

[37] Ernie Brickell, Gary Graunke, Jean-Pierre Seifert, *Mitigating cache/timing attacks in AES and RSA software implementations*, RSA Conference 2006, San Jose, session DEV-203, 2006, `http://2006.rsaconference.com/us/cd_pdfs/DEV-203.pdf`

[38] R. Briol, *Emanation: how to keep your data confidential*, proc. Symposium on Electromagnetic Security for Information Protection, SEPI'91, Rome, Italy, 1991

[39] J. P. Buhler, Hendrik W. Lenstra, Jr., Carl Pomerance, *Factoring integers with the Number Field Sieve*, 50–94 in [129], 1993

[40] E. R. Canfield, Paul Erdös, C. Pomerance, *On a problem of Oppenheim concerning "Factorisatio Numerorum"*, J. Number Theory, vol. 17, 1–28, 1983

[41] Anne Canteaut, Cédric Lauradoux, André Seznec, *Understanding cache attacks*, research report RR-5881, INRIA, April 2006, `http://www-rocq.inria.fr/codes/Anne.Canteaut/Publications/RR-5881.pdf`

[42] Eugène Olivier E. Carissan, *Machine a resoudre les congruences*, Bulletin de la Société d'Encouragement pour l'Industrie Nationale, vol. 132, 600-607, 1920

[43] Stefania Cavallar, *Strategies in filtering in the Number Field Sieve*, proc. Algorithmic Number Theory, 4th International Symposium (ANTS-IV), 209–232, 2000

[44] Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Walter Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, Paul Leyland, Joël Marchand, François Morain, Alec Muffett, Chris Putnam, Craig Putnam, Paul Zimmermann, *Factorization of a 512-bit RSA modulus*, proc. Eurocrypt 2000, LNCS 1807, 1–17, Springer, 2000

[45] Li Chen, Wayne Eberly, Erich Kaltofen, B. David Saunders, Willam J. Turner, Gilles Villard, *Efficient matrix preconditioners for black box linear algebra*, Linear Algebra and its Applications, vol. 343–344, 119–146, 2002

[46] Scott Contini, Arjen K. Lenstra, Ron Steinfeld, *VSH, an efficient and provable collision resistant hash function*, proc. Eurocrypt 2006, Springer, LNCS 4004, 165–182, 2006

[47] Don Coppersmith, *Modifications to the number field sieve*, Journal of Cryptology, vol. 169–180, 1993

[48] Don Coppersmith, *Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm*, Mathematics of Computation, vol. 62, 333–350, 1994

[49] Don Coppersmith, *Solving homogeneous linear equations over* $GF(2)$ *via block Wiedemann algorithm*, Mathematics of Computation, vol. 62 issue 205, 333–350, 1994

[50] Jean-Marc Couveignes, *Computing a square root for the Number Field Sieve*, 95–102 in [129], 1993

[51] R. Crandall, Carl Pomerance, *Prime numbers: a computational perspective*, Springer-Verlag, 2001

[52] Joan Daemen, Vincent Rijmen, *AES Proposal: Rijndael*, version 2, AES submission, 1999, `http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf`

[53] Nicolaas Govert de Bruijn, *On the number of positive integers $\leq x$ and free of prime factors $> y$, II*, Indagationes Mathematicae, vol. 38, 239–247, 1966

[54] Pierre de Fermat, *Fragment d'une lettre de Fermat*, Oeuvres de Fermat 2, 256–258, 1894

[55] Leonard Eugene Dickson, *History of the theory of numbers, vol. I: Divisibility and Primality*, Carnegie Institute of Washington, Publication No. 256, 1919

[56] *Discrete logarithms in $GF(p)$ using the number field sieve*, SIAM Journal on Discrete Mathematics, vol. 6 issue. 1, 124–138, 1993

[57] Cynthia Dwork, Andrew Goldberg, Moni Naor, *On memory-bound functions for fighting spam*, proc. 2003, 426–444, LNCS 2729, Springer, 2003

[58] Cynthia Dwork, Moni Naor, Omer Reingold, *Immunizing encryption schemes from decryption errors*, proc. Eurocrypt 2004, LNCS 3027, 342–360, Springer, 2004

[59] Electronic Frontier Foundation, *DES Cracker Project*, web site, `http://www.eff.org/descracker.html`

[60] L. B. Evans, H. E. Bass., *Tables of absorption and velocity of sound in still air at* 68°," Wyle Laboratories, Report WR72-2, 1972

[61] Uriel Feige, Prabhakar Raghavan, *Exact analysis of hot-potato routing* proc. 33rd Annual IEEE Symp. on Foundations of Computer Science (FOCS), 553–562, IEEE, 1992

[62] Jens Franke et al., *RSA576*, e-mail announcement, December 2003, `http://www.loria.fr/~zimmerma/records/rsa576`

[63] Jens Franke, *On the factorization of RSA200*, invited talk at Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'06), 2006, `http://www.hyperelliptic.org/tanja/SHARCS/talks06/Jens_Franke.pdf`

[64] Jens Franke, Thorsten Kleinjung, C program for Number Field Sieve polynomial selection, private communication, Feb. 2003

[65] Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, Colin Stahlke, *SHARK — a realizable special hardware device for factoring 1024-bit integers*, proc. Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'05), 2005

[66] Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, Colin Stahlke, *SHARK — a realizable special hardware device for factoring 1024-bit integers*, proc. Cryptographic Hardware and Embedded Systems (CHES) 2005, LNCS 3659, Springer, 2005

[67] Free Software Foundation Inc., *The GNU Privacy Guard*, web site, `http://gnupg.org`

[68] Carl Friedrich Gauss, *Disquisitiones arithmeticae*, Braunschweig, 1801

[69] Willi Geiselmann, Fabian Januszewski, Hubert Köpfer, Jan Pelzl, Rainer Steinwandt, *A simpler sieving device: combining ECM and TWIRL*, proc. International Conference on Information and Communications Security (ICICS) 2006, LNCS 4296, Springer, 2006

[70] Willi Geiselmann, Hubert Köpfer, Rainer Steinwandt, Eran Tromer, *Improved routing-based linear algebra for the Number Field Sieve*, proc. International Conference on Information Technology: Coding and Computing (ITCC'05), vol. 1, 636-641, IEEE, 2005

[71] Willi Geiselmann, Adi Shamir, Rainer Steinwandt, Eran Tromer, *A systolic design for supporting Wiedemann's algorithm*, proc. Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'05), 13–17, 2005

[72] Willi Geiselmann, Adi Shamir, Rainer Steinwandt, Eran Tromer, *Scalable hardware for sparse systems of linear equations, with applications to integer factorization*, proc. Cryptographic Hardware and Embedded Systems (CHES) 2005, LNCS 3659, 131–146, Springer, 2005

[73] Willi Geiselmann, Adi Shamir, Rainer Steinwandt, Eran Tromer, *Fault-tolerance in hardware for sparse systems of linear equations, with applications to integer factorization*, Chapter 8 in N. Nedjah, L. de Macedo Mourelle (Eds.), New Trends in Cryptographic Systems, Nova Science Publishers, 2006

[74] Willi Geiselmann, Rainer Steinwandt, *A dedicated sieving hardware*, proc. PKC 2003, LNCS 2567, 254–266, Springer-Verlag, 2003

[75] Willi Geiselmann, Rainer Steinwandt, *Hardware to solve sparse systems of linear equations over GF(2)*, proc. Cryptographic Hardware and Embedded Systems (CHES) 2003, LNCS 2779, 51–63, Springer, 2003

[76] Willi Geiselmann, Rainer Steinwandt, *Yet another sieving device*, proc. CT-RSA 2004, LNCS 2964, 278–291, Springer, 2004

[77] Willi Geiselmann, Rainer Steinwandt, *Non-wafer-scale sieving hardware for the NFS: another attempt to cope with 1024-bit*, IACR Cryptology ePrint Archive, report 2006/403, 2006, `http://eprint.iacr.org/2006/403`

[78] Genesis 27:5

[79] GNU Project, *GNU Scientific Library (GSL)*, `http://www.gnu.org/software/gsl`

[80] Oded Goldreich, *Foundations of Cryptography - Volume 1*, Cambridge University Press, 2001

[81] Oded Goldreich, *Foundations of Cryptography - Volume 2*, Cambridge University Press, 2004

[82] Oded Goldreich, Shafi Goldwasser, Shai Halevi, *Public-key cryptosystems from lattice reduction problems*, proc. CRYPTO '97, LNCS 1294, 112–131, Springer, 1997

[83] Oded Goldreich, Rafail Ostrovsky, *Software protection and simulation on oblivious RAMs*, Journal of the ACM, vol. 43 no. 3, 431–473, 1996

[84] M. D. Grammatikakis, D. F. Hsu, M. Kraetzl, J. F. Sibeyn, *Packet routing in fixed-connection networks: a survey*, Journal of Parallel and Distributed Computing, vol. 54 no. 2, 77–132, 1998

[85] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman, *NTRU: A Ring-Based Public Key Cryptosystem*, proc. Algorithmic Number Theory (ANTS III), LNCS 1423, 267–288, Springer, 1998

[86] Wei-Ming Hu, *Reducing timing channels with fuzzy time*, proc. IEEE Computer Society Symposium on Research in Security and Privacy, 8–20, IEEE, 1991

[87] Wei-Ming Hu, *Lattice scheduling and covert channels*, IEEE Symposium on Security and Privacy, 52–61, IEEE, 1992

[88] IEEE, *IEEE Std. 1363a-2004 IEEE Standard Specifications for Public-Key Cryptography – Amendment 1: Additional Techniques*, 2004

[89] D. Ierardi, *2d-bubblesorting in average time* $O(N\lg N)$, Proceedings 6th ACM symposium on Parallel algorithms and architectures, 1994

[90] Russell Impagliazzo, *A personal view of average-case complexity*, proc. Structure in Complexity Theory Conference, 134–147, IEEE, 1995

[91] International Technology Roadmap for Semiconductors, *International Technology Roadmap for Semiconductors 2001 Edition*, `http://www.itrs.net/reports.html`

[92] International Technology Roadmap for Semiconductors, *International Technology Roadmap for Semiconductors 2002 Update*, `http://www.itrs.net/reports.html`

[93] International Technology Roadmap for Semiconductors, *International Technology Roadmap for Semiconductors 2003 Edition*, `http://www.itrs.net/reports.html`

[94] William Stanley Jevons, *The principles of science: a treatise on logic and scientific method*, vol. I, Macmillan and Co., London, 1874

[95] Jonah 3:3

[96] Christos Kaklamanis, Danny Krizanc, Satish Rao, *Hot-potato routing on processor arrays*, ACM Symposium on Parallel Algorithms and Architectures archive, 273–282, ACM, 1993

[97] Burt Kaliski, *TWIRL and RSA key size*, RSA Laboratories Technical Note, `http://www.rsasecurity.com/rsalabs/node.asp?id=2004`, 2003

[98] Erich Kaltofen, *Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems*, , Mathematics of Computation, vol. 64 no. 210, 777–806, 1995

[99] Erich Kaltofen, Austin A. Lobo, *Distributed matrix-free solution of large sparse linear systems over finite fields*, Algorithmica, vol. 24 no. 3–4, 331–348, 1999

[100] J. Kelsey, B. Schneier, D. Wagner, C. Hall, *Side channel cryptanalysis of product ciphers* (final version); `http://www.schneier.com/paper-side-channel2.pdf`

[101] John Kelsey, Bruce Schneier, David Wagner, Chris Hall, *Side channel cryptanalysis of product ciphers*, proc. 5th European Symposium on Research in Computer Security, LNCS 1485, 97–110, Springer, 1998

[102] Stephen Kent et al., *RFC 4301* through *RFC 4309*, Network Working Group Request for Comments, `http://rfc.net/rfc4301.html` etc., 2005

[103] Richard E. Kessler, Mark D. Hill, *Page placement algorithms for large real-indexed caches*, ACM Transactions on Computer systems, vol, 10, no., 4, 338–359, 1992

[104] Hea Joung Kim, William H. Magione-Smith, *Factoring large numbers with programmable hardware*, proc. FPGA 2000, ACM, 2000

[105] Thorsten Kleinjung, private communication, May 2003

[106] Thorsten Kleinjung, *Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024 bit integers*, Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'06), 2006

[107] Donald E. Knuth, Luis Trabb Pardo, *Analysis of a simple factorization algorithm*, Theoretical Computer Science, vol. 3, issue 3, 321–348, 1976

[108] Paul Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*, proc. CRYPTO 96, LNCS 1109, 104–113, Springer, 1996.

[109] Paul Kocher, Joshua Jaffe, Benjamin Jun, *Differential Power Analysis*, proc. CRYPTO 99, LNCS 1666, 388–397, Springer Verlag, 1999

[110] François Koeune, Jean-Jacques Quisquater, *A timing attack against Rijndael*, technical report CG-1999/1, Université catholique de Louvain, `http://www.dice.ucl.ac.be/crypto/tech_reports/CG1999_1.ps.gz`

[111] Markus G. Kuhn, *Compromising emanations: eavesdropping risks of computer displays*, Technical Report UCAM-CL-TR-577, University of Cambridge, Computer Laboratory, 2003

[112] Mirosław Kutyłowski, Krzysztof Loryś, Brigitte Oesterdiekhoff, Rolf Wanka, *Fast and feasible periodic sorting networks of constant depth*, proc. proc. 35th Annual IEEE Symp. on Foundations of Computer Science (FOCS), 369–380, IEEE, 1994

[113] Robert Lambert, *Computational aspects of discrete logarithms*, Ph.D. University of Waterloo, 1996

[114] Robert Lambert, private communication, Sep. 2003

[115] Cédric Lauradoux, *Collision attacks on processors with cache and countermeasures*, Western European Workshop on Research in Cryptology (WEWoRC) 2005, Lectures Notes in Informatics, vol. P-74, 76–85, 2005, `http://www.cosic.esat.kuleuven.ac.be/WeWorc/allAbstracts.pdf`

[116] Frederick William Lawrence, *Factorisation of numbers*. Quarterly Journal of Pure and Applied Mathematics, vol. 28, 285–311, 1896

[117] Derrick H. Lehmer, *The mechanical combination of linear forms*, The American Mathematical Monthly, vol. 35, 114–121, 1928

[118] Derrick H. Lehmer, *Hunting big game in the theory of numbers*, Scripta Mathematica I, 229–235, Sept. 1932

[119] Derrick H. Lehmer, *A photo-electric number sieve*, American Mathematical Monthly, vol. 40, 401–406, 1933

[120] Derrick H. Lehmer, *The sieve problem for all-purpose computers*, Mathematical Tables and Other Aids to Computation, vol. 7, no. 41, 6–14, 1953

[121] Derrick H. Lehmer, *An announcement concerning the delay line SIEVE DLS-127*, Mathathatics of Computation, vol. 20, 645-646, 1966

[122] Derrick H. Lehmer, *Exploitation of parallelism in number theoretic and combinatorial computation*, proc. 5th Manitoba Conf. on Numerical Mathematics and Computing, 95–111, 1976

[123] Derrick H. Lehmer, *A history of the sieve process*, in *A History of Computing in the Twentieth Century*, 445–456, Academic Press, 1980

[124] F. T. Leighton, F. Makedon, I. Tollis, *A* $2n-2$ *step algorithm for routing in an* $n \times n$ *mesh*, proc. 1989 ACM Symposium on Parallel Algorithms and Architectures, 328–335, ACM, 1989

[125] Arjen K. Lenstra, *Integer factoring*, Designs, Codes and Cryptography, vol. 19, 101-128, 2000

[126] Arjen K. Lenstra, private communication, Apr. 2006

[127] Arjen K. Lenstra, Bruce Dodson, *NFS with four large primes: an explosive experiment*, proc. Crypto '95, LNCS 963, 372–385, Springer, 1995

[128] Arjen K. Lenstra, Hendrik W. Lenstra, Mark Manasse, John M. Pollard, *The factorization of the ninth Fermat number*, Mathematics of Computation, vol. 61, 318–349, 1993

[129] Arjen K. Lenstra, H. W. Lenstra, Jr. (eds.), *The development of the number field sieve*, Lecture Notes in Mathematics, vol. 1554, Springer, 1993

[130] Arjen K. Lenstra, Adi Shamir, *Analysis and Optimization of the TWINKLE Factoring Device*, proc. Eurocrypt 2002, LNCS 1807, 35–52, Springer, 2000

[131] Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, Eran Tromer, *Analysis of Bernstein's factorization circuit*, proc. Asiacrypt 2002, LNCS 2501, 1–26, Springer, 2002

[132] Arjen K. Lenstra, Eran Tromer, Adi Shamir, Wil Kortsmit, Bruce Dodson, James Hughes, Paul Leyland, *Factoring estimates for a 1024-bit RSA modulus*, procedings of Asiacrypt 2003, Springer, LNCS 2894, 331–346, Springer, 2003

[133] Tim Lindholm, Frank Yellin, *The Java virtual machine specification*, 2nd edition, Prentice Hall, 1999

[134] Seth Lloyd, *Ultimate physical limits to computation*, Nature, vol. 406, 1047-1054, 2000

[135] J. L. Massey, *Shift-register synthesis and BCH decoding*, IEEE Transactions on Information Theory, vol. 15, 122–127, 1969

[136] Joel McNamara, *The complete, unofficial TEMPEST information page*, web page, `http://www.eskimo.com/~joelm/tempest.html`, 2004

[137] Thomas S. Messerges, *Using second-order power analysis to attack DPA resistant software*, proc. Cryptographic Hardware and Embedded Systems (CHES) 2000, Springer, LNCS 1965, 238–251, 2000

[138] Thomas S. Messerges, Ezzy A. Dabbish, Robert H. Sloan, *Power analysis attacks of modular exponentiation in smartcards*, proc. Cryptographic Hardware and Embedded Systems (CHES) 1999, Springer, LNCS 1717, 144–157, 1999

[139] Robert V. Meushaw, Mark S. Schneider, Donald N. Simard, Grant M. Wagner, *Device for and method of secure computing using virtual machines*, US patent 6,922,774, 2005

[140] Microsoft Corp., *Next-generation secure computing base*, web page, `http://www.microsoft.com/resources/ngscb`

[141] Gary L. Miller, *Riemann's Hypothesis and tests for primality*, Journal of Computer and System Sciences 13, no. 3, 300-317, 1976

[142] Peter L. Montgomery, *A block Lanczos algorithm for finding dependencies over GF(2)*, proc. Eurocrypt'95, LNCS 925, 106–120, Springer-Verlag 1995

[143] Peter L. Montgomery, B. Murphy, *Improved polynomial selection for the number field sieve*, extended abstract for the conference on the mathematics of public-key cryptography, June 13-17, 1999, The Fields institute, Toronto, Ontario, Canada

[144] Pieter Moree, *On the psixyology of Diophantine equations*, Ph.D. thesis, Leiden University, 1993

[145] Brian Murphy, *Modelling the yield of the Number Field Sieve polynomials*, Proceedings ANTS-III, LNCS 1423, 137–151, Springer, 1998

[146] Brian Murphy, *Polynomial selection for the Number Field Sieve integer factorisation algorithm*, Ph.D. thesis, The Australian National University, July 1999

[147] Nahum 1:8

[148] Moni Naor, Guy N. Rothblum, *The complexity of online memory checking*, proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS) 2005, 573–584, IEEE 2005

[149] National Institute of Standards and Technology, *Advanced Encryption Standard (AES)*, FIPS PUB 197, 2001

[150] National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, FIPS PUB 180-2, 2002

[151] National Institute of Standards and Technology, *Key management guidelines, Part 1: General guidance (draft)*, Jan. 2003

[152] National Institute of Standards and Technology, *Recommendation for key management — Part 1: General (Revised)*, NIST Special Publication 800-57, May 2006, `http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf`

[153] Michael Neve, Jean-Pierre Seifert, *Advances on access-driven cache attacks on AES*, proc. Selected Areas in Cryptography (SAC'06), to appear.

[154] Michael Neve, Jean-Pierre Seifert, Zhenghong Wang, *A refined look at Bernstein's AES side-channel analysis*, proc. ACM Symposium on Information, computer and communications security, 369–369, 2006

[155] Phong Nguyen, *A Montgomery-like square root for the Number Field Sieve*, proc. Algorithmic Number Theory (ANTS-III), LNCS 1423, 151–168, Springer, 1998

[156] A. M. Odlyzko, *Discrete logarithms: the past and the future*, Designs, Codes and Cryptography, 129–145, Springer-Verlag, 2000

[157] OpenVPN Solutions LLC, *OpenVPN — An Open Source SSL VPN Solution by James Yonan*, web site, `http://openvpn.net`

[158] Yossi Oren, Adi Shamir, *Power analysis of RFID tags*, RSA Conference 2006 panel session; see `http://www.wisdom.weizmann.ac.il/~yossio/rfid`

[159] H. Orman, P. Hoffman, *RFC 3766: Determining strengths for public keys used for exchanging symmetric keys*, Network Working Group Request for Comments, `http://rfc.net/rfc3766.html`, 2004

[160] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, Vincent Rijmen, *A side-channel analysis resistant description of the AES S-box*, proc. Fast Software Encryption (FSE) 2005, LNCS 3557, Springer, 2005

[161] Daniel Page, *Theoretical use of cache memory as a cryptanalytic side-channel*, technical report CSTR-02-003, Department of Computer Science, University of Bristol, 2002, `http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=1000625`

[162] Daniel Page, *Defending against cache-based side-channel attacks*, Information Security Technial Report, vol. 8 issue. 8, 2003

[163] Daniel Page, *Partitioned cache architecture as a side-channel defence mechanism*, IACR Cryptology ePrint Archive, report 2005/280, 2005, `http://eprint.iacr.org/2005/280`

[164] Pascal Paillier, *Public-key cryptosystems based on composite degree residuosity classes*, proc. Eurocrypt 99,, LNCS 1592, 223-238, Springer, 1999

[165] Cameron D. Patterson, *The derivation of a high speed sieve device*, Ph.D. thesis, Dept. of Computer Science, University of Calgary, Calgary, Alberta, Canada, 1991

[166] O. Penninga, *Finding column dependencies in sparse matrices over mathbbF_2 by block Wiedemann*, Technical report MAS-R9819, Centrum voor Wiskunde en Informatica, 1998

[167] Colin Percival, *Cache missing for fun and profit*, BSDCan 2005, Ottawa, 2005; see `http://www.daemonology.net/hyperthreading-considered-harmful`

[168] Carl Pomerance, *A Tale of Two Sieves*, Notices of the AMS, 1473–1485, Dec. 1996

[169] Carl Pomerance, Jeffrey W. Smith, Randy Tuler, *A pipeline architecture for factoring large integers with the quadratic sieve algorithm*, SIAM Journal on Computing, vol. 17, 387–403, 1988

[170] Carl Pomerance, Jeffrey W. Smith, Samuel S. Wagstaff, Jr., *New ideas for factoring large integers*, proc. CRYPTO'83, Plenum Press, 81–85, 1984

[171] , John Proos, *Imperfect decryption and an attack on the NTRU encryption scheme*, IACR Cryptology ePrint Archive, report 2003/002, 2003

[172] Michael O. Rabin, *Digitalized signatures and public-key functions as intractable as factorization*, MIT Lab for Computer Science technical report LCS/TR212, 1979

[173] Michael O. Rabin, *Probabilistic algorithm for testing primality*, Journal of Number Theory 12, no. 1, 128-138, 1980

[174] Oded Regev, *Lattice-based cryptography*, tutorial given in CRYPTO 2006, `http://www.cs.tau.ac.il/~odedr/papers/crypto2006.pdf`

[175] Regulatory Authority for Telecommunications and Posts, *Suitable cryptographic algorithms*, German Federal Gazette, no. 48, 4202–4203, 11 March 2003, `http://www.bundesnetzagentur.de/media/archive/1873.pdf`

[176] Ronald L. Rivest, Adi Shamir, Leonard Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM, vol. 21 no. 2, 120–126. 1978

[177] RSA Security Inc., *The RSA factoring challenge FAQ*, `http://www.rsasecurity.com/rsalabs/challenges/factoring` and *The RSA factoring challenge numbers*, `http://www.rsasecurity.com/rsalabs/challenges/factoring/numbers.html`

[178] W. G. Rudd, Duncan A. Buell, Donald M. Chiarulli, *A high performance factoring machine*, proc. 11th International Conference on Computer Architecture, 297-300, ACM, 1983

[179] Ruhr-Universität Bochum Chair for Communication Security, *Side Channel Cryptanalysis Lounge*, web site, `http://www.crypto.ruhr-uni-bochum.de/en_sclounge.html`

[180] Manfred Schimmler, *Fast sorting on the instruction systolic array*, Report 8709, Christian Albrecht University Kiel, 1987

[181] Werner Schindler, *A timing attack against RSA with the Chinese Remainder Theorem*, proc. Cryptographic Hardware and Embedded Systems (CHES) 2000, LNCS 1965, 109–124, Springer, 2000

[182] O. Schirokauer, *Discrete logarithms and local units*, Philosophical Transactions of the Royal Society of London, A 345, 409–423, 1993

[183] Claus P. Schnorr, Adi Shamir, *An optimal sorting algorithm for mesh connected computers*, proc. 16th ACM Symposium on Theory of Computing, 255–263, 1986

[184] Jeffrey Shallit, Hugh C. Williams, François Morain, *Discovery of a lost factoring machine*, The Mathematical Intelligencer, vol. 17 no. 3, 41–47, 1995

[185] Adi Shamir, *Factoring large numbers with the TWINKLE device (extended abstract)*, proc. Cryptographic Hardware and Embedded Systems (CHES) 1999, LNCS 1717, 2–12, Springer, 1999

[186] Adi Shamir, Dag Arne Osvik, Eran Tromer, *Cache attacks and countermeasures: the case of AES*, proc. RSA Conference Cryptographers Track (CT-RSA) 2006, LNCS 3860, 1–20, Springer, 2006; first presented at the FSE'05 rump session, February 2005

[187] Adi Shamir, Eran Tromer, *Factoring large numbers with the TWIRL device*, proc. CRYPTO 2003, LNCS 2729, 1–26, Springer, 2003

[188] Adi Shamir, Eran Tromer, *On the cost of factoring RSA-1024*, RSA CryptoBytes, vol. 6 no. 2, 10–19, 2003

[189] Adi Shamir, Eran Tromer, *Acoustic cryptanalysis: on nosy people and noisy machines*, Eurocrypt 2004 rump session, 2004; see [191]

[190] Adi Shamir, Eran Tromer, *Special-purpose hardware for factoring: the NFS sieving step*, proc. Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'05), 1–12, 2005

[191] Adi Shamir, Eran Tromer, *Acoustic cryptanalysis: on nosy people and noisy machines*, web page, `http://tromer.org/acoustic`

[192] Peter W. Shor, *Algorithms for quantum computation: Discrete logarithms and factoring*, proc. 35th Annual Symposium on the Foundations of Computer Science, 124–134, IEEE, 1994

[193] Peter W. Shor, *Polynomial-time algorithm for prime factorization and discrete logarithms on a quantum computer*, SIAM Journal on Computing, vol. 26 no. 5, 1484–1509, 1997

[194] Jop F. Sibeyn, *Overview of mesh results*, Technical Report MPI-95-1018, Max-Planck Institut fur Informatik, Germany, 1995

[195] SigBlips, *baudline signal analyzer*, web site, `http://baudline.com`

[196] Robert D. Silverman, *Optimal parameterization of SNFS*, Manuscript, 2002, `http://citeseer.ist.psu.edu/silverman03optimal.html`

[197] Robert D. Silverman, *A cost-based security analysis of symmetric and asymmetric key lengths*, Bulletin 13, RSA Security, 2000, `http://www.rsasecurity.com/rsalabs/node.asp?id=2088`

[198] Jeffrey W. Smith, Samuel S. Wagstaff, Jr., *An extended precision operand computer*, proc. 21st Southeast Region. ACM Conference, 209–216, 1983

[199] Robert M. Solovay, Volker Strassen, *A fast Monte-Carlo test for primality*, SIAM Journal on Computing, vol. 6 no. 1, 84–85, 1977

[200] Allan J. Stephens, *OASiS: An open architecture sieve system for problems in number theory*, Ph.D. thesis, Department of Computer Science, University of Manitoba, 1990

[201] Herman te Riele, Stefania Cavallar, Bruce Dodson, Arjen Lenstra, Paul Leyland, Walter Lioen, Peter Montgomery, Brian Murphy, Paul Zimmermann, *Factorization of RSA-140 using the Number Field Sieve*, e-mail announcement, February 1999, `http://ftp.cwi.nl/herman/NFSrecords/RSA-140`

[202] Eran Tromer, *Clockwise Transposition Routing examples*, web page, `http://tromer.org/clockwise`

[203] Eran Tromer, *Special-purpose cryptanalytic devices: an annotated taxonomy*, web page, `http://tromer.org/cryptodev`

[204] Trusted Computing Group, *Trusted Computing Group: Home*, web site, `http://www.trustedcomputinggroup.org`

[205] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, Hiroshi Miyauchi, *Cryptanalysis of DES implemented on computers with cache*, proc. Cryptographic Hardware and Embedded Systems (CHES) 2003, LNCS 2779, 62-76, 2003

[206] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, Hiroshi Miyauchi, *Cryptanalysis of block ciphers implemented on computers with cache*, proc. International Symposium on Information Theory and its Applications 2002, 803–806, 2002

[207] Yukiyasu Tsunoo, Etsuko Tsujihara, Maki Shigeri, Hiroyasu Kubo, Kazuhiko Minematsu, *Improving cache attacks by considering cipher structure, International Journal of Information Security*, "Online First", Springer, Nov. 2005

[208] University of Cambridge Computer Laboratory, *The Xen virtual machine monitor*, web site, `http://www.cl.cam.ac.uk/research/srg/netos/xen`

[209] Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood, Isaac L. Chuang, *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance*, Nature, vol. 414, 883–887, 2001

[210] Alexander A. Veith, Andrei V. Belenko, Alexei Zhukov, *A preview on branch misprediction attacks: using Pentium performance counters to reduce the complexity of timing atttacks*, CRYPTO'06 rump session, 2006

[211] Gilles Villard, *A study of Coppersmith's block Wiedemann algorithm using matrix polynomials*, Rapport de Recherche 975 IM, Institut d'Informatique et de Mathematiques Appliquees de Grenoble, France, 1997. Full version of [212].

[212] Gilles Villard, *Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems (extended abstract)*, proc. 1997 International Symposium on Symbolic and Algebraic Computation, 32–39, ACM Press, 1997. Extended abstract of [211].

[213] Gilles Villard, *Block solution of sparse linear systems over* GF(q)*: the singular case*, ACM SIGSAM Bulletin, vol. 32 issue. 4, 10–12, ACM, 1998

[214] VMware Inc., *VMware: virtualization, virtual machine & virtual server consolidation*, web site, `http://www.vmware.com`

[215] B. Weis, *RFC 4359: The use of RSA/SHA-1 signatures within encapsulating security payload (ESP) and authentication header (AH)*, Network Working Group Request for Comments, `http://rfc.net/rfc4359.html`, 2006

[216] D. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory, vol. IT-32, 54–62, 1986

[217] Michael J. Wiener, *The full cost of cryptanalytic attacks*, Journal of Cryptology, vol. 17 no. 2, 105–124, 2004

[218] Hugh C. Williams, Jeffrey O. Shallit, *Factoring integers before computers*, 481–531 in *Mathathatics of Computation 1943–1993: a half-century of computational mathematics*, AMS, 1994

[219] Wolfram Research Inc., *Mathematica*, `http://www.wolfram.com/products/mathematica`

[220] Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'05), Feb. 24–25, Paris, France, 2005, `http://www.ruhr-uni-bochum.de/itsc/tanja/SHARCS/slides.html`

[221] Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'06), April 3–4, Cologne, Germany, 2006, `http://www.ruhr-uni-bochum.de/itsc/tanja/SHARCS/start06.html`

[222] Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'07), Sep. 9–10, Vienna, Austria, 2006, `http://sharcs.org`

[223] Peter Wright, *Spycatcher*, Viking Penguin, 1987

[224] Li Zhuang, Feng Zhou, J. D. Tygar, *Keyboard acoustic emanations revisited*, proc. 12th ACM Conference on Computer and Communications Security, 373–382, 2005

[225] Xiaotong Zhuang, Tao Zhang, Santosh Pande, *HIDE: An Infrastructure for Efficiently protecting information leakage on the address bus*, proc. Architectural Support for Programming Languages and Operating Systems, 82–84, ACM, 2004

[226] Xiatong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, Santosh Pande, *Hardware assisted control flow obfuscation for embedded processors*, proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 292-302, ACM, 2004

Index of notation

The following points to the primary definition(s) of the non-standard notation used in this text. Note that, where no confusion can arise, a symbol may carry different meanings in different chapters.

Symbols

L

Index