Special-Purpose Hardware for Factoring: the NFS Sieving Step

Adi Shamir Eran Tromer

Weizmann Institute of Science
{shamir, tromer}@wisdom.weizmann.ac.il

Abstract

In the quest for factorization of larger integers, the present bottleneck is the sieving step of the Number Field Sieve algorithm. Several special-purpose hardware architectures have been proposed for this step: TWINKLE (based on electro-optics), mesh circuits (based on two-dimensional systolic arrays) and TWIRL (based on parallel processing pipelines). For 1024-bit composites, the use of such special-purpose hardware has reduced the predicted cost of factorization by 5-6 orders of magnitude. We review the proposed architectures, their cost, and their various approaches to exploiting the flexibility of custom hardware.

1 Introduction

The hardness of factoring large integers drawn from appropriate distributions is a central assumption in cryptography, and underlies many public-key cryptosystems and protocols. The most efficient algorithm known for factoring large integers is the Number Field Sieve (NFS) algorithm [12]. Thus, barring theoretical breakthroughs, the security of cryptosystems such as RSA practically relies on the feasibility of the NFS algorithms for the relevant input sizes.

The time and space complexities of NFS are subexponential in the size of the input composite, though the analysis is heuristic in parts and leaves considerable uncertainty in regard to concrete estimates. Several concrete factorization experiments have been performed, most notably the factorization of a 512-bit RSA key in 1999 [4] and the factorization of a 576-bit RSA key in 2003 [1]. Extrapolating from such experiments, it was predicted that factoring 1024-bit composites by similar means would presently require trillions of

Invited talk at the Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS) 2005, February 2005.

dollars. Consequentially, it was often claimed that 1024-bit RSA keys are safe for the next 15 to 20 years (e.g., [3, 21] and a NIST guideline draft [17]).

However, while the above experiments have employed large workstation clusters and Cray supercomputers, they have always used general-purpose computer hardware. When the workload is sufficiently high (either because the composites are large or because there are many of them to factor), there arises the prospect of employing special-purpose hardware that is tailored to the task.

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 example of this approach is the EFF DES Cracker [6], 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 beyond efficient implementation of standard algorithms. Custom circuit designs allows for specialized data paths, flexible partitioning of resources, enormous parallelism, and even for the use non-electric physical phenomena. Taking advantage of these requires new algorithms or adaptation of existing ones.[1]

Such special-purpose devices have been proposed for two steps of the NFS algorithms, which together dominate its cost:

The sieving step. The goal of this step is to find many "relations", i.e., integers of a certain form that have only small prime divisors. Hardware devices for sieve-like algorithms date back to Lehmer's bicycle chain device [13] from 1926, but the first modern proposal in the context of the Number Field Sieve is the TWINKLE device [19, 15]. Subsequent proposals include mesh-based sieving [8, 10] and TWIRL [20].

The linear algebra (matrix) step. The goal of this step is to find linear relations among columns of a large but sparse matrix over GF(2), whose columns represent the relations found in the sieving step or combinations thereof. Special-purpose hardware for this was devised by Bernstein [2] (and inspired some of the aforementioned sieving devices). It was subsequently analyzed and improved in several works [16, 9, 7, 11].

At present, for 1024-bit and using the most efficient architectures proposed, the sieving step is more expensive than the linear algebra step even if the NFS parameters are optimized for minimal sieving cost.[2] Thus, for estimating the total cost of factoring 1024-bit integers, the sieving step discussed here is the prime concern. In the following, we describe the sieving step, and survey the aforementioned special-purpose hardware architectures devised for it.

[1] Several upper and lower bounds for arbitrary physical devices were given by Wiener [22].

[2] See [16], but note that the specific cost estimates mentioned there are superseded by the above works.

2 Preliminaries

2.1 The NFS Sieving Step

We begin by briefly reviewing the NFS sieving step, after significant simplification of non-essential details (for an introduction to the NFS algorithm the reader is referred to [18], and for a detailed account, to [12]). For simplicity, we assume the use of line sieving.

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* B. There is, on average, one pair per such prime, and thus roughly $B/\ln B$ pairs in total. Each pair (p_i, r_i) corresponds to an arithmetic progression $P_i = \{a : a \equiv r_i \pmod{p_i}\}$. We are interested in identifying the sieve locations $a \in \{0, \ldots, R-1\}$ that are members of many progressions P_i with large p_i:

$$g(a) > T \quad \text{where} \quad g(x) = \sum_{i:a \in P_i} \log_h p_i$$

for some small constant h. It is permissible to have "small" errors in this threshold check; in particular, we round all logarithms to the nearest integer. For each a that exceeds the threshold, we also need to find the set $\{i : a \in P_i\}$ of progressions that contribute to $g(a)$. Out of the H sieve locations, only $(6/\pi^2)H$ on average are potentially useful and the rest can be ignored.

We need to perform $2H$ such sieving tasks, partitioned into H instances of the *rational sieve* with $B = B_{\mathrm{r}}$, and H instances of the *algebraic sieve* with $B = B_{\mathrm{a}}$, where generally the latter is more expensive since $B_{\mathrm{a}} > B_{\mathrm{r}}$.[3]

1024-bit parameters. For concreteness, we mention (and occasionally assume) the following choice of parameters for 1024-bit composites: $B_{\mathrm{r}} = 3.5 \cdot 10^9$, $B_{\mathrm{a}} = 2.6 \cdot 10^{10}$, $R = 1.1 \cdot 10^{15}$, and $H = 2.7 \cdot 10^8$. Thus, each of the two sieves inspects a total of $(6/\pi^2) \cdot R \cdot H \approx 1.8 \cdot 10^{23}$ sieve locations.[4]

2.2 Traditional Sieving

The traditional method of performing the sieving task is a variant of Eratosthenes's sieve algorithm for finding primes. 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 $\log_h p_i$ 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 $\log_h p_i$ contribution is low.

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

[4] The choice of parameters in NFS has several degrees of freedom, and is not fully understood other than asymptotically. This specific choice was analyzed in [14] and assumed for TWIRL in [20].

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. 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 B then most progressions make very few contributions (if any) to each chunk, so the amortized cost per contribution increases. Thus, a large amount of memory is required, both for the accumulators and for storing the input (that is, the list of progressions). As Bernstein [2] 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 each [15]. In today's prices and assuming a fivefold improvement in the relevant PC performance criteria since [15], this would cost about US$ 13M.[5]

Cost for 1024-bit composites. The cost of traditional sieving for 1024-bit composites is prohibitive, as shown 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 $(6/\pi^2) \cdot R \cdot H \cdot (7+3) \cdot 1\text{ns} \approx 57$ million years. 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 1 year would be over US$ 10^{11} with these parameters.[6] This lower bound is consistent with predication by extrapolation [21], which yields cost on the order of US$ 10^{12} in current terms.

3 TWINKLE

The TWINKLE sieving device [19, 15] consists of a wafer containing numerous independent electronic 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 $\log_h p_i$ iff $a \in P_i$. The values emitted at each clock cycle are summed to obtain $g(a)$, 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 is proportional to $\log p_i$. A light sensor above the wafer measures the total light intensity in each clock cycle, which

[5] A similar figure can be obtained analogously to the next paragraph, using the concrete line-sieving parameters from [20, 14] instead of the extrapolated special-q sieving used in [15].

[6] A different NFS parameter choice may somewhat reduce the cost, as may the use of special-q lattice sieving [12], but neither is expected to dramatic increase the feasibility.

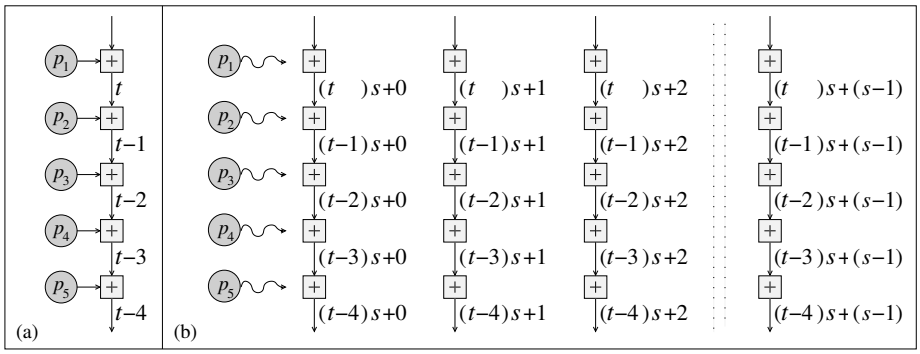

Figure 1: Flow of sieve locations through the device in (a) pipeline-of-adders TWINKLE (b) TWIRL.

is proportional to $g(a)$, and reports a success when this exceeds a given threshold. The cells themselves are implemented by simple registers and ripple adders. To support the optoelectronic operations, it was originally proposed to use 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 [19, 15].

Compared to traditional sieving, TWINKLE exchanges the roles of space and time:

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

Pipeline-of-adders TWINKLE. A different variant of the TWINKLE device replaces the electro-optics in with standard electronic circuits.[7] This variant also operates at a rate of one sieve location per clock cycle, but does so using a pipelined systolic chain of electronic adders. It consists of a long unidirectional bus, $\log_2 h$ 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 $\log_h p_i$ 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 $g(0)$, followed by $g(1), \ldots, g(R)$, one per clock cycle. See Fig. 1(a).

Cost for 768-bit composites. The cost of TWINKLE for 768-bit composites was analyzed in [15]. It is 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

[7] This variant of TWINKLE was considered in [15], and deemed inferior in that context. We recall it here as a predecessor (and a suboptimal special case) of TWIRL.

prices, and assuming a fivefold increase since [15] 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 enough cells to represent all the progressions), and these auxiliary computers turn out to form a bottleneck.

4 TWIRL

4.1 Approach

The TWIRL sieving device [20] is a purely electronic device, which follows the time-space reversal of TWINKLE but increases the throughput by simultaneously processing thousands of sieve locations at each clock cycle. Since this is done without duplication of the input progressions (i.e., the pairs (p_i, r_i)), the cost per unit of throughput decreases dramatically. Equivalently stated, the cost of storing the huge input is amortized across many parallel processes. Moreover, TWIRL uses an extremely compact representation of the input progressions; specifically, it stores them in DRAM memory (as opposed to the registers employed by prior designs) while ensuring a purely sequentially access pattern to avoid large memory access latencies.

Achieving the above requires a more complex architecture, whose main ideas will be sketched in the following by starting with the pipeline-of-adders TWINKLE described above and applying a series of changes. For concreteness, throughout this section we will concentrate on the rational sieve of 1024-bit factorization, with the parameters given in Section 2.1 and the corresponding optimized TWIRL parameters from [20].

Compared to the pipeline-of-adders TWINKLE, TWIRL's parallelization is obtained by handling the sieve range $\{0, \ldots, R-1\}$ in consecutive chunks of length $s = 4096$.[8] To do so, the bus is thickened by a factor of s and now contains s logical lines, where each line carries 10-bit numbers. 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 $\{g(0), \ldots, g(s-1)\}$; they appear simultaneously, followed by successive disjoint groups of size s, one group per clock cycle. See Fig. 1(b).

We now have to add the $\log_h p_i$ contributions to all s lines in parallel. Obviously, the naive solution of duplicating all the adders s times gains nothing in terms of equipment cost per unit of throughput. If we try to use the TWINKLE-like circuitry without duplication, we encounter difficulties in scheduling and communicating the contributions across the thick bus: the sieve locations flow down the bus (in Fig. 1(b), vertically), and the contributions

[8] $s = 4096$ applies to the rational sieve. For the algebraic sieve (see Section 4.3) we use even higher parallelism: $s = 32,768$.

should somehow travel across the bus (horizontally) and reach an appropriate adder at exactly the right time.

Accordingly, the simple TWINKLE-like cells are replaced by other units that perform scheduling and routing. Each such unit, called a *station*, handles some small portion of the progressions; its interface consists of bus input, bus output, clock and some circuitry for loading the inputs. Many stations are connected serially in a pipeline, and at the end of the pipeline (i.e., at the output of the last station) there lies a threshold check unit that produces the device output.

While the purpose of all the stations is identical, the device employs a heterogeneous architecture that contains three different station designs — the progression intervals p_i come in a very large range of sizes, and different sizes involve very different design tradeoffs. The progressions are partitioned into stations according to the size of their intervals p_i, and the optimal station design is employed in each case. The most important station design is that used for the "largish primes" — those above some threshold (here, $5.2 \cdot 10^5 < p_i < B$). It is described in the following.

4.2 Largish primes

For every prime smaller than $B = 3.5 \cdot 10^9$ there is (on average) one progression. Thus the majority of progressions have intervals p_i that are much larger than $s = 4096$, so they produce $\log_h p_i$ contributions very seldom. For 1024-bit composites there is a huge number (about $1.6 \cdot 10^8$) of such progressions; even with TWINKLE's simple emitter cells, these cannot be fitted into a single silicon wafer. The primary consideration is thus to store these progressions as compactly as possible, while maintaining a low cost per contribution. An important characteristic of this station design is that it holds its progressions in compact DRAM-type memory using only sequential (and thus very efficient) read/write access. This efficient allows fitting 4 independent 1024-bit TWIRL devices (each of which is $s = 4096$ times faster than TWINKLE) into a single 30cm silicon wafer using well-established chip manufacturing technology (e.g., 130nm process).

The station design for these progressions is shown in Fig. 2 (after some simplifications). The progressions are partitioned into $8,490$ memory banks, so that each bank contains many progressions. Each progression is stored in one of these memory banks, where at any given time it is represented by an *event* of the form (p_i, ℓ_i, τ_i), whose meaning is: "`at time` τ_i`, send a` $\log_h p_i$ `contribution to bus line` ℓ_i."

Each memory bank is connected to a special-purpose processor, which continuously processes these events and sends corresponding *emissions* of the form "`add` $\log_h p_i$ `to bus line` ℓ_i" to attached delivery lines, which span the bus. Each delivery line acts as a shift register that carries the emissions across the bus. Additionally, at every intersection between a delivery line and a bus line there is a conditional adder; when the emission reaches its destination bus line ℓ_i, the value $\log_h p_i$ is added to the value that passes through that point of the bus pipeline at that moment.

Thus, sieve locations are (logically) flowing down the bus at a constant velocity, and

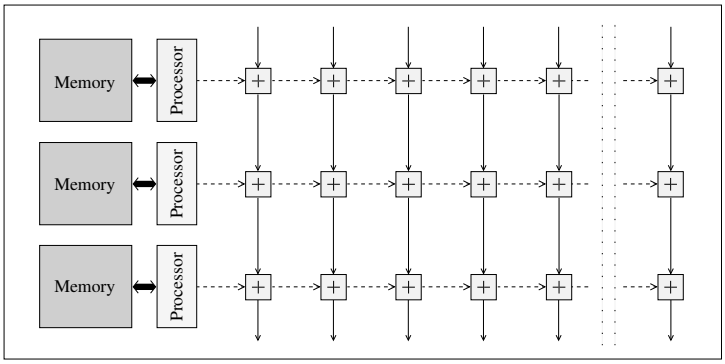

Figure 2: Schematic structure of a (simplified) largish station.

emissions are being sent across the bus at a constant velocity. To ensure that each emission "hits" its target at the right time, the two perpendicular flows must be perfectly synchronized, which requires a lot of care. However, the benefit is that the cost per contribution is very low: most of the time the event is stored very compactly in the form of an event in DRAM; then, for a brief moment it occupies the processor, and finally it occupies a delivery line for the minimum possible duration — the amount of time needed to travel across the bus to the destination bus line.

It is the processor's job to ensure accurate scheduling of emissions.[9] The ideal way to achieve this would be to store the events in a priority queue that is sorted by the emission time τ_i. Then, the processor would simply repeat the following loop:[10]

1. Pop the next event (p_i, ℓ_i, τ_i) from the priority queue.
2. Wait until time τ_i and then send an emission to the delivery line, addressed to bus line ℓ_i.
3. Compute the next event (p_i, ℓ_i', τ_i') of this progression, and push it into the priority queue.

Standard implementations of priority queues (e.g., the heap data structure) are unsuitable for our purposes, due to the passive nature of standard DRAM and high latency. First, the processor would need to make a logarithmic number of memory accesses at each iteration. Worse yet, these memory accesses occur at unpredictable places, and thus incur a high random-access overhead. This is addressed by the following.

Sequential-access priority queue. By taking advantage of the unique properties of the sieving problem we can get a good approximation of a priority queue that is highly

[9] In the full design [20], there is an additional component, called a *buffer*, which performs fine-tuning and load balancing.

[10] For simplicity, here we ignore the possibility of collisions.

efficient. Briefly, the idea is as follows. The events are read sequentially from memory (step 1 above) in a cyclic order, at constant rate. When the new calculated event is written back to memory (step 3 above), it is written to a memory address that will be read just before its schedule time τ_i'. Since both τ_i' and the read schedule are known, this memory address is easily calculated by the processor. In this way, after a short stabilization period the processor always reads imminent events,[11] exactly as desired. Each iteration now involves just one sequential-access read operation and one random-access write operation. In addition, it turns out that with appropriate choice of parameters we can cause the write operations to always occur in a small window of activity, just behind the "read head". We may thus view the $8,490$ memory banks as closed rings of various sizes, with an active window "twirling" around each ring at a constant linear velocity. Each such sliding window is handled by a fast SRAM-based cache, whose content is swapped in and out of DRAM in large blocks. This allows the bulk of events to be held in DRAM. Better yet, now the only interface to the DRAM memory is through the SRAM cache; this allows elimination of various peripheral circuits that are needed in standard DRAM.

4.3 Other Highlights

Other station designs. For progressions with small interval ($p_i < 5.2 \cdot 10^5$), it is inefficient to continuously shuttle the progression state to and from passive memory. Thus, each progression is handled by an independent active *emitter* cell that includes an internal counter (similarly to TWINKLE). An emitter serves multiple bus lines, using a variant of the delivery lines described above. Using certain algebraic tricks, these cells can be made very compact. Two such station designs are used: for the progressions with medium-sized intervals, many progressions share the same delivery lines (since emissions are still not very frequent); this requires some coordination logic. For very small intervals, each emitter cell has its own delivery line.

Diaries. Recall that in addition to finding the sieve locations a whose contributions exceed the threshold, we also want to find the sets $\{i : a \in P_i\}$ of relevant progressions. This is accomplished by adding a *diary* to each processor (it suffices to handle the progressions with large interval). The diary is a memory bank which records every emission sent by the processor and saves it for a few thousand clock cycles — the depth of the bus pipeline. By that time, the corresponding sieve location a has reached the end of the bus and the accumulated sum of logarithms $g(a)$ was checked. If the threshold was exceeded, this is reported to all processors and the corresponding diary entries are recalled and collected. Otherwise, these diary entries are discarded (i.e., their memory is reused).

Cascading the sieves. Recall that in NFS perform two sieving tasks in parallel: a *rational sieve* whose parameters were given above, and an *algebraic sieve* which is usually more expensive since it has a large value of B. However, we succeed in greatly reducing the cost of the algebraic sieve by using an even higher parallelization factor for it: $s = 32,768$.

[11] Collisions are handled by adding appropriate slacks.

This is made possible by an alteration that greatly reduces the bus width: the algebraic sieve needs only to consider the sieve locations that passed the rational sieve, i.e., about one in 5,000. Thus we connect the input of the algebraic sieve to the output of the rational sieve, and in the algebraic sieve we replace the thick bus and delivery lines by units that consider only the sieve locations that passed the rational sieve. We now have a much narrower bus containing only 32 lines, though each line now carries both a partial sum (as before) and the index a of the sieve location to which the sum belongs. Logically, the sieve locations still travel in chunks of size s, so that the regular and predictable timing is preserved. Physically, only the "relevant" locations (at most 32) in each chunk are present; emissions addressed to the rest are discarded.

Fault tolerance. The issue of fault tolerance is very important, as silicon wafers normally have multiple local faults, and in a wafer-scale device one cannot simply discard faulty chips. For 1024-bit composites TWIRL is a wafer-scale design, and is designed to operate in the presence of faults by a combination of methods, such as routing around faults and re-assigning the functionality of faulty units to spare ones. Note that occasional transient faults can be tolerated, since in the sieving task only the total number of good a values matters.

4.4 Cost

Cost for 1024-bit composites. The cost of constructing enough TWIRL devices to factor a 1024-bit integer in 1 year has been estimated at US$ 1.1M [14], when using the now-standard 90nm chip manufacturing process. The non-recurring R&D cost would be on the order of US$ 10M–20M. For the older 130nm process technology, the cost and performance of TWIRL has been estimated in more detail in [20], with a bottom line of US$ 10M for completion in 1 year. The following contains some key points of the latter case.

Recall that to implement NFS we have to perform two different sieving tasks, a rational sieve and an algebraic sieve, which have different parameters. Here, the rational sieve (whose parameters were given above) dominates the cost. For this sieve, we can fit 4 complete TWIRL devices on a 30cm silicon wafer. For the algebraic sieve (in the cascaded variant sketched above), a TWIRL device occupies a full wafer, but uses a higher parallelization factor, $s = 32,768$. For both types of TWIRL devices, the circuit area is occupied by the DRAM banks storing large progressions. These devices are assembled in clusters that consist of 8 rational TWIRL devices (occupying two wafers) and 1 algebraic TWIRL (on a third wafer), where each rational TWIRL has a unidirectional link to the algebraic TWIRL over which it transmits 12 bits per clock cycle. At a 1GHz clock rate, the sieving task can be completed by 194 independent TWIRL clusters running in parallel.

Cost for 768-bit composites. For 768-bit composites, a single silicon wafer containing 6 independent TWIRL clusters (using 130nm process) can complete the sieving in 95 days. Alas, the non-recurring cost would still be on the order of US$ 10M–20M.

Notes. 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).

5 Mesh-based sieving

An alternative approach to highly parallel sieving hardware, which predated (and inspired) TWIRL, is the mesh-based sieving proposed by Geiselmann and Steinwandt [8]. It followed a related proposal by Bernstein [2], and was subsequently improved in [10]. Here we only sketch the basic idea in its simplest form, and refer the reader to [8, 10] further details and significant improvements.

The device consists of a two-dimensional mesh of $s \times s$ nodes, with each node connected only to its (at most) 4 neighbors.[12] 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 corresponding packet; the packet specifies the corresponding contribution $\log_h p_i$, and the address of the mesh node in charge of sieve location a. Second, 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 $\log_h p_i$ to its accumulator and discards the packet. Thus, once all 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.

Improvements. Some of the improvement on the above described in [8, 10] include the following. Progressions with large intervals are represented using compact DRAM storage. Progressions with small intervals produce most of the contributions, but there are relatively few of them, so their representation is duplicated across the mesh to avoid routing their contributions over long distances. The routing network uses the heuristic

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

clockwise transposition algorithm [16] to minimize cost. The topology is changed from a mesh to a torus, or several overlapping tori.

Cost for 768-bit composites. The mesh-based design of [10] has an estimated cost marginally higher than that of TWIRL. For the same throughput, its silicon area is 6.3 times larger than that quoted for 768-bit TWIRL in [20], but it is divided into smaller independent chips (which is practically advantageous); if TWIRL is re-parametrized to use similarly sized chips, the gap is roughly halved.

Notes. While their architectures are radically different, the the mesh-based design and TWIRL are fundamentally strongly related: both use high parallelism to amortize the cost of the progression storage over many simultaneously processed sieve locations, both handle primes of different sizes by different method (and specifically use compact DRAM storage for larger primes), and both use a two-dimensional layout data movement. Compared to traditional sieving, both asymptotically reduce the area×time cost by a factor of $s = \tilde{\Theta}(\sqrt{B})$ using a device of size $s^2 = \tilde{\Theta}(B)$. Also, both are bandwidth-intensive, and thus limited by the maximal wafer size (or by inter-chip I/O throughput). The tradeoffs are fairly subtle: the approach of [10] enables a compressed encoding of progression in DRAM, whereas TWIRL uses smaller and faster logic to achieve higher parallelization and speed. One advantage of the mesh-based approach is the use of a uniform chip design whose repeating units are smaller than those of TWIRL, which simplifies design and manufacturing.

6 Conclusions

It has been often claimed that 1024-bit RSA keys are safe for the next 15 to 20 years, since when applying the Number Field Sieve to such composites both the sieving step and the linear algebra step would be unfeasible. However, this assumed implementation using standard general-purpose computers. The introduction of special-purpose hardware architectures for NFS has reduced the predicted cost of factoring 1024-bit integers by several orders of magnitude, to within the reach of large organizations. Focusing on the sieving step of NFS, we have surveyed the main proposed architectures.

References

[1] J. Franke et al, *RSA576*, e-mail announcement, Dec. 2003, `http://www.loria.fr/~zimmerma/records/rsa576`

[2] Daniel J. Bernstein, *Circuits for integer factorization: a proposal*, manuscript, 2001,`http://cr.yp.to/papers.html`

[3] Richard P. Brent, *Recent progress and prospects for integer factorisation algorithms*, proc. COCOON 2000, LNCS 1858, 3–22, Springer-Verlag, 2000

[4] S. Cavallar, B. Dodson, A.K. Lenstra, W. Lioen, P.L. Montgomery, B. Murphy, H.J.J. te Riele, et al., *Factorization of a 512-bit RSA modulus*, proc. Eurocrypt 2000, LNCS 1807, 1–17, Springer-Verlag, 2000

[5] Don Coppersmith, *Modifications to the number field sieve*, Journal of Cryptology, vol. 6, 169–180, 1993

[6] Electronic Frontier Foundation, *DES Cracker Project*, `http://www.eff.org/descracker.html`

[7] Willi Geiselmann, Hubert Köpfer, Rainer Steinwandt, Eran Tromer, *Improved Routing-Based Linear Algebra for the Number Field Sieve*, proc. International Conference on Information Technology (ITCC) '05 – Track on Embedded Cryptographic Systems, IEEE, 2005

[8] Willi Geiselmann, Rainer Steinwandt, *A dedicated sieving hardware.*, proc. Public Key Cryptography — PKC 2003, LNCS 2567, 254–266. Springer-Verlag, 2003

[9] Willi Geiselmann, Rainer Steinwandt. *Hardware for solving sparse systems of linear equations over GF(2)*, proc. Cryptographic Hardware and Embedded Systems — CHES 2003, LNCS 2779, 51–61, Springer-Verlag, 2003

[10] Willi Geiselmann, Rainer Steinwandt, *Yet another sieving device*, CT-RSA 2004, LNCS 2964,278–291, Springer, 2004

[11] Willi Geiselmann, Adi Shamir, Rainer Steinwandt, Eran Tromer, *A systolic design for supporting Wiedemann's algorithm*, presented at Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS) workshop, Paris, 2005

[12] Arjen K. Lenstra, H.W. Lenstra, Jr., (eds.), *The development of the number field sieve*, Lecture Notes in Math., vol. 1554, Springer-Verlag, 1993

[13] D. H. Lehmer, *The mechanical combination of linear forms*, The American Mathematical Monthly, vol. 35, 114–121, 1928

[14] Arjen K. Lenstra, Eran Tromer, Adi Shamir, Wil Kortsmit, Bruce Dodson, James Hughes, Paul Leyland, *Factoring estimates for 1024-bit RSA modulus*, proc. Asiacrypt 2003, LNCS 2894, 331–346, Springer-Verlag, 2003 2003, LNCS, Springer-Verlag, to appear.

[15] Arjen K. Lenstra, Adi Shamir, *Analysis and optimization of the TWINKLE factoring device*, proc. Eurocrypt 2002, LNCS 1807, 35–52, Springer-Verlag, 2000

[16] Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, Eran Tromer, *Analysis of Bernstein's factorization circuit*, proc. Asiacrypt 2002, LNCS 2501, 1–26, Springer-Verlag, 2002

[17] NIST, *Key management guidelines, Part 1: General guidance (draft)*, Jan. 2003, `http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html`

[18] Carl Pomerance, *A Tale of Two Sieves*, Notices of the AMS, 1473–1485, Dec. 1996

[19] Adi Shamir, *Factoring large numbers with the TWINKLE device (extended abstract)*, proc. CHES'99, LNCS 1717, 2–12, Springer-Verlag, 1999

[20] Adi Shamir, Eran Tromer, *Factoring large numbers with the TWIRL device*, proc. Crypto 2003, LNCS 2729, Springer-Verlag, 2003

[21] Robert D. Silverman, *A cost-based security analysis of symmetric and asymmetric key lengths*, Bulletin 13, RSA Security, 2000

[22] Michael J. Wiener, *The Full Cost of Cryptanalytic Attacks*, Journal of Cryptology, vol. 17 no. 2, 105–124, 2004