On the Cost of Factoring RSA-1024

Adi Shamir Eran Tromer

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

Abstract

As many cryptographic schemes rely on the hardness of integer factorization, exploration of the concrete costs of factoring large integers is of considerable interest. Most research has focused on PC-based implementations of factoring algorithms; these have successfully factored 530-bit integers, but practically cannot scale much further. Recent works have placed the bottleneck at the sieving step of the Number Field Sieve algorithm. We present a new implementation of this step, based on a custom-built hardware device that achieves a very high level of parallelism "for free". The design combines algorithmic and technological aspects: by devising algorithms that take advantage of certain tradeoffs in chip manufacturing technology, efficiency is increased by many orders of magnitude compared to previous proposals. Using this hypothetical device (and ignoring the initial R&D costs), it appears possible to break a 1024-bit RSA key in one year using a device whose cost is about $10M (previous predictions were in the trillions of dollars).

1 Introduction

The security of many cryptographic schemes and protocols depends on the hardness of finding the factors of large integers drawn from an appropriate distribution. The best known algorithm for factoring large integers is the Number Field Sieve (NFS)[1], whose time and space complexities are subexponential in the size of the composite. However, little is known about the real complexity of this problem. The evident confidence in the hardness of factoring comes from observing that despite enormous interest, no efficient factoring algorithm has been found.

To determine what key sizes are appropriate for a given application, one needs concrete estimates for the cost of factoring integers of various sizes. Predicting these costs has proved notoriously difficult, for two reasons. First, the performance of modern factoring algorithms is not understood very well: their complexity analysis is often asymptotic and heuristic, and leaves large uncertainty factors. Second, even when the exact algorithmic complexity is known, it is hard to estimate the concrete cost of a suitable hypothetical large-scale computational effort using current technology; it's even harder to predict what this cost would be at the end of the key's planned lifetime, perhaps a decade or two into the future.

Due to these difficulties, common practice is to rely on extrapolations from past factorization experiments. Many such experiments have been performed and published; for example, the successful factorization of a 512-bit RSA key in 1999 [5] clearly indicated

[1] See [10] for the seminal works and [17] for an introduction. The subtask we discuss is defined in Section 2.1.

the insecurity of such keys for many applications, and prompted a transition to 1024-bit keys (often necessitating software or hardware upgrades).[2] The current factorization record, obtained nearly four years later in March 2003, stands at 530 bits [1].[3] From this data, and in light of the subexponential complexity of the algorithm used, it seems reasonable to surmise that factoring 1024-bit RSA keys, which are currently in common use, should remain infeasible for well over a decade.

However, the above does not reflect a fundamental economy-of-scale consideration. While the published experiments have employed hundreds of workstations and Cray supercomputers, they have always used general-purpose computer hardware. However, when the workload is sufficiently high (either because the composites are large or because there are many of them to factor), it becomes more efficient to construct and employ custom-built hardware dedicated to the task. Direct hardware implementation of algorithms is considerably more efficient than software implementations, and makes it possible to eliminate the expensive yet irrelevant peripheral hardware found in general-purpose computers. An example of this approach is the EFF DES Cracker [7], built in 1998 at a cost of $210,000 and capable of breaking a DES key in expected time of 4.5 days using 36864 search units packed into 1536 custom-built gate array chips. Indeed, its equipment cost per unit of throughput was much lower than similar experiments that used general-purpose computers.

Custom-built hardware can go beyond efficient implementation of standard algorithms — it allow specialized data paths, enormous parallelism and can even use non-electronic physical phenomena. Taking advantage of these requires new algorithms or adaptation of existing ones. One example is the TWINKLE device [18, 13], which implements the sieving step of the NFS factoring algorithm using a combination of highly parallel electronics and an analog optical adder.

Recently, D. J. Bernstein made an important observation [3] about the major algorithmic steps in the NFS algorithm. These steps have a huge input, which is accessed over and over many times. Thus, traditional PC-based implementations are very inefficient in their use of storage: a huge number of storage bits is just sitting in memory, waiting for a single processor to access them. Most of the previous work on NFS cost analysis (with the notable exception of [21]) considered only the number of processor instructions, which is misleading because the cost of memory greatly outweighs the cost of the processor. Instead, one should consider the equipment cost per unit of throughput, i.e., the construction cost multiplied by the running time per unit of work.

Following this observation, Bernstein presented a new parallel algorithm for the matrix step of the NFS algorithm, based on a mesh-connected array of processors. Intuitively, the idea is to attach a simple processor to each block of memory and execute a distributed algorithm among these processors to get better utilization of the memory. With this algorithm, and by changing some adjustable parameter in the NFS algorithm so as to minimize "cost per unit of throughput" rather than instruction count, Bernstein's algorithm allows one to factor integers that are 3.01 times longer compared to traditional algorithms (though only 1.17 times longer when the traditional algorithms are re-optimized for throughput cost). Subsequent works [14, 8] evaluated the practicality of Bernstein's algorithm for 1024-bit composites, and suggested improved versions that significantly reduced its cost. With these hypothetical (but detailed) designs, the cost of the matrix step was brought down from trillions of dollars [21] to at most a few dozen million dollars (all figures are for completing the task in 1 year).

This left open the issue of the other major step in the Number Field Sieve, namely the sieving step.

[2] Earlier extrapolations indeed warned of this prospect.

[3] Better results were obtained for composites of a special form, using algorithms which are not applicable to RSA keys.

For 1024-bit composites it was predicted that sieving would require trillions of dollars,[21][4] and would be impractical even when using the TWINKLE device. This article discusses a new design for a custom-hardware implementation of the sieving step, which reduces this cost to about \$10M. The new device, called TWIRL[5], can be seen as an extension of the TWINKLE device. However, unlike TWINKLE it does not have optoelectronic components, and can thus be manufactured using standard VLSI technology on silicon wafers. The underlying idea is to use a single copy of the input to solve many subproblems in parallel. Since input storage dominates cost, if the parallelization overhead is kept low then the resulting speedup is obtained essentially for free. Indeed, the main challenge lies in achieving this parallelism efficiently while allowing compact storage of the input. Addressing this involves myriad considerations, ranging from number theory to VLSI technology. The resulting design is sketched in the following sections, and a more detailed description appears in [19].

2 Context

2.1 The Sieving Task

The TWIRL device is specialized to a particular task, namely the sieving task which occurs in the Number Field Sieve (and also in its predecessor, the Quadratic Sieve). This section briefly reviews the sieving problem, with many simplifications.

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. 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 $g(a)$.

We shall concentrate on 1024-bit composites and a particular choice of the adjustable NFS parameters, with $R = 1.1 \cdot 10^{15}$ and $B = 3.5 \cdot 10^9$. We need to perform $H = 2.7 \cdot 10^8$ such sieving tasks, called *sieve lines*, that have different (though related) inputs.[6] The numerical values that appear below refer to this specific parameter choice.

2.2 Traditional Sieving

The traditional method of performing the sieving task is a variant of Eratosthenes's 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 $C[a] > T$. The point is that 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.

When this algorithm is implemented on a PC, we cannot apply it to the full range $a = 0,\ldots,R-1$ since

[4] [15] gave a lower bound of about \$160M for a one-day effort. This disregarded memory, but is much closer to our results since the new device greatly reduces the amortized cost of memory.

[5] TWIRL stands for The Weizmann Institute Relation Locator.

[6] In fact, for each sieve line we need to perform two sieves: a "rational sieve" and an "algebraic sieve" (see Section 3.3). The parameters given here correspond to the rational sieve, which is responsible for most (two thirds) of the device's cost.

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 [3] observed, this is inherently inefficient because each memory bit is accessed very infrequently.

2.3 Sieving with TWINKLE

An alternative way of performing the sieving was proposed in the TWINKLE device [18, 13], which operates as follows. 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 $\{0 \leq a < R\}$. 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 $g(x)$, 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 using analog optics: 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, 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, TWINKLE uses Gallium Arsenide wafers (alas, these are relatively small, expensive and hard to manufacture compared to silicon wafers, which are readily available). Compared to traditional sieving, TWINKLE exchanges the roles of space and time:

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

3 TWIRL

3.1 Approach

The TWIRL device 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 with (almost) no duplication of the input, the equipment cost per unit of throughput decreases dramatically. Equivalently, we can say that the cost of storing the huge input is amortized across many parallel processes.

As a first step toward TWIRL, consider an electronic variant of TWINKLE which still operates at a rate of one sieve location per clock cycle, but does so using a pipelined systolic chain of electronic adders.[7] Such a device would consist of a long unidirectional bus, 10 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).

The 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 $(t-z)s+i$,

[7] This variant was considered in [13], but deemed inferior in that context.

[8] $s = 4096$ applies to the rational sieve. For the algebraic sieve (see Section 3.3) we use even higher parallelism: $s = 32768$.

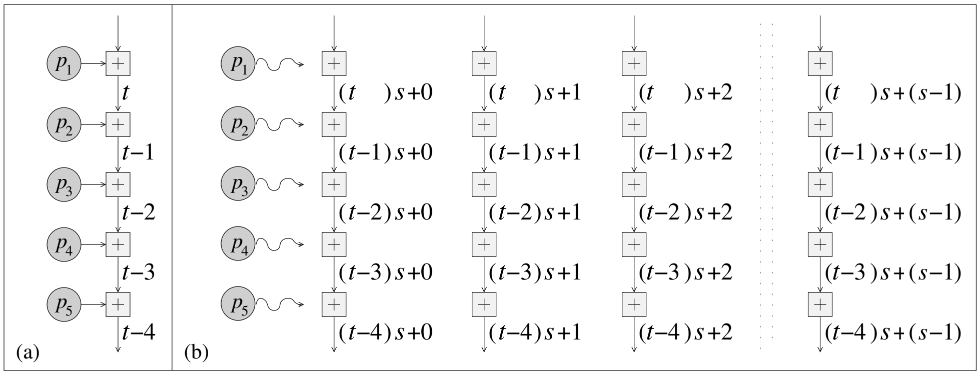

Figure 1: Flow of sieve locations through the device in (a) a chain of adders and (b) TWIRL.

$i \in \{0, \ldots, s-1\}$. 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 should somehow travel across the bus (horizontally) and reach an appropriate adder at exactly the right time.

Accordingly, we replace the simple TWINKLE-like cells 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. 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.

While the function of all the stations is identical, we use a heterogeneous architecture that employs three different station designs — the 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.

Due to space limitations, we describe only the most important station design, which is used for the majority of progressions. The other station designs, and additional details, can be found in [19].

3.2 Large 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, we could not fit all of them into a single wafer. The primary consideration is thus to store these progressions as compactly as possible, while maintaining a low cost per contribution. Indeed, we will succeed in storing these progressions in compact DRAM-type memory using only sequential (and thus very efficient) read/write access. This necessitates additional support logic, but its cost is amortized across many progressions. This efficient storage lets us fit 4 independent 1024-bit TWIRL devices (each of which is $s = 4096$ times faster than TWINKLE) into a single 30cm silicon wafer.

The station design for these progressions (namely, those with $p_i > 5.2 \cdot 10^5$) is shown in Fig. 2 (after some simplifications). The progressions are partitioned into $8{,}490$ memory banks, so that each bank contains many (between 32 and $2.2 \cdot 10^5$) 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[9]; 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 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.[10] 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:[11]

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 significant random-access overhead. Fortunately, by taking advantage of the unique properties of the sieving problem we can get a good approximation of a priority queue that is highly 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

[9]We use carry-save adders, which are very compact and have low latency (the tradeoff is that the bus lines now use a redundant representation of the sums, which doubles the bit-width of the bus).

[10]In the full design [19], there is an additional component, called a *buffer*, which performs fine-tuning and load balancing.

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

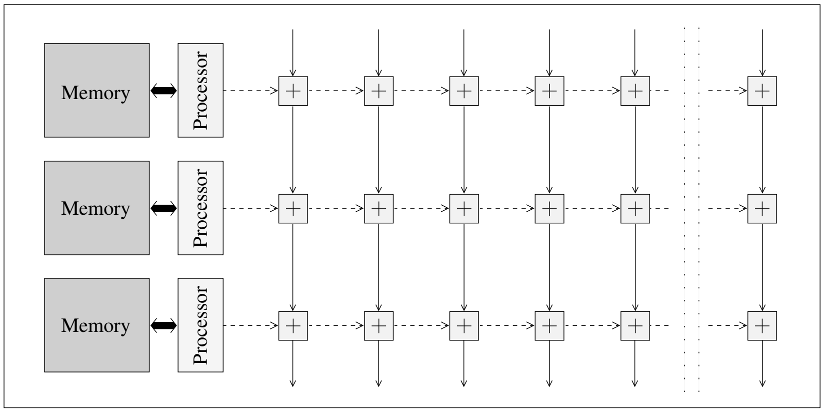

Figure 2: Schematic structure of a (simplified) largish station.

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,[12] 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.

[12]Collisions are handled by adding appropriate slacks.

3.3 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. In the Number Field Sieve we have to 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: $s = 32{,}768$. 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. When the wafer contains many independent small chips, one usually discards the faulty ones. However, for 1024-bit composites TWIRL is a wafer-scale design and thus must operate in the presence of faults. All large components of TWIRL can be made fault-tolerant by a combination of techniques: routing around faults, post-processing and re-assigning faulty units to spare. We can tolerate occasional transient faults since the sieving task allows a few errors; only the total number of good a values matters.

4 Cost

Based on the detailed design, we estimated the cost and performance of the TWIRL device using today's VLSI technology (namely, the 0.13μm process used in many modern memory chips and CPUs). While these estimates are hypothetical, they rely on a detailed analysis and should reasonably reflect the real cost. It should be stressed that the NFS parameters assumed are partially based on heuristic estimates. See [19] for details.

1024-bit composites. 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, a TWIRL device requires $15{,}960\text{mm}^2$ of silicon wafer area, so we can fit 4 of them on a 30cm silicon wafer. Most of the device area is occupied by the large progressions (and specifically, 37% of the device is used for their DRAM banks). For the algebraic sieves we use a higher parallelization factor, $s = 32{,}768$. One algebraic TWIRL device requires $65{,}900\text{mm}^2$ of silicon wafer area — a full wafer — and here too most of the device is occupied by the largish progressions (the DRAM banks occupy 66%).

The devices are assembled in clusters that consist of 8 rational TWIRLs (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. A cluster handles a full sieve line in $R/32{,}768$ clock cycles, i.e., 33.4 seconds when clocked at 1GHz. The full sieving involves H sieve lines, which would require 194 years when using a single cluster (after a heuristic that rules out 33% of the sieve locations). 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,[13] it appears realistic that all the sieving required for factoring 1024-bit integers can be completed within 1 year by a device that costs $\$10\mathrm{M}$ to manufacture. In addition to this per-device cost, there would be an initial NRE cost on the order of $\$20\mathrm{M}$ (for design, simulation, mask creation, etc.).

512-bit composites. Since 512-bit factorization is well-studied [18, 13, 8] and backed by experimental data [5], it is interesting to compare 512-bit TWIRL to previous designs. We shall use the same 512-bit parameters as in [13, 8], though they are far from optimal for TWIRL. With $s = 1{,}024$, we can fit 79 TWIRLs into a single silicon wafer; together, they would handle a sieve line in 0.00022 seconds (compared to 1.8 seconds for TWINKLE wafer and 0.36 seconds for a full wafer using mesh-based design of [8]). Thus, in factoring 512-bit composites the basic TWIRL design is about $1{,}600$ times more cost effective than the best previously published design [8], and $8{,}100$ times more cost effective than TWINKLE. Such a wafer full of TWIRLs, which can be manufactured for about $5,000 in large quantities, can complete the sieving for 512-bit composites in under 10 minutes (this is before TWIRL-specific optimizations which would halve the cost, and using the standard but suboptimal parameter choice).

768-bit composites. For 768-bit composites, a single wafer containing 6 TWIRL clusters can complete the sieving in 95 days. This wafer would cost about $5,000 to manufacture — one tenth of the RSA-768 challenge prize [20]. Unfortunately these figures are not easy to verify experimentally, nor do they provide a quick way to gain $45,000, since the initial NRE cost remains $10M-$20M.

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

5 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 (e.g., [4, 21] and a NIST guideline draft [16]). However, these estimates relied on PC-based implementations. We presented a new design for a custom-built hardware implementation of the sieving step, which relies on algorithms that are highly tuned for the available technology. With appropriate settings of the NFS parameters, this design reduces the cost of sieving to about $10M (plus a one-time cost of $20M). Recent works [14, 9] indicate that for these NFS parameters, the cost of the matrix step is even lower.

Our estimates are hypothetical and rely on numerous approximations; the only way to learn the precise costs involved would be to perform a factorization experiment. 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. This should be taken into account by anyone planning to use a 1024-bit RSA key.

Acknowledgment. This work was inspired by Daniel J. Bernstein's insightful work on the NFS matrix step, and its adaptation to sieving by Willi Geiselmann and Rainer Steinwandt. We thank the latter for interesting discussions of their design and for suggesting an improvement to ours. We are indebted to Arjen K. Lenstra for many insightful discussions, and to Robert D. Silverman, Andrew "bunnie" Huang, Michael Szydlo and Markus Jakobsson for valuable comments and suggestions. Early versions of [12] and the polynomial selection programs of Jens Franke and Thorsten Kleinjung were indispensable in obtaining refined estimates for the NFS parameters.

References

[1] F. Bahr, J. Franke, T. Kleinjung, M. Lochter, M. Böhm, *RSA-160*, e-mail announcement, Apr. 2003, `http://www.loria.fr/~zimmerma/records/rsa160`

[2] Daniel J. Bernstein, *How to find small factors of integers*, manuscript, 2000, `http://cr.yp.to/papers.html`

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

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

[5] 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

[6] Don Coppersmith, *Modifications to the number field sieve*, Journal of Cryptology, **6**(3) 169–180, 1993

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

[8] Willi Geiselmann, Rainer Steinwandt, *A dedicated sieving hardware*, proc. PKC 2003, LNCS **2567** 254–266, Springer-Verlag, 2002

[9] Willi Geiselmann, Rainer Steinwandt, *Hardware to solve sparse systems of linear equations over GF(2)*, proc. CHES 2003, LNCS, Springer-Verlag, to appear.

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

[11] Arjen K. Lenstra, Bruce Dodson, *NFS with four large primes: an explosive experiment*, proc. Crypto '95, LNCS **963** 372–385, Springer-Verlag, 1995

[12] Arjen K. Lenstra, Bruce Dodson, James Hughes, Wil Kortsmit, Paul Leyland, *Factoring estimates for a 1024-bit RSA modulus*, proc. Asiacrypt 2003, LNCS, Springer-Verlag, to appear.

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

[14] 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

[15] Arjen K. Lenstra, Eric R. Verheul, *Selecting cryptographic key sizes*, Journal of Cryptology, **14**(4) 255–293, 2002

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

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

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

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

[20] RSA Security, *The new RSA factoring challenge*, web page, Jan. 2003, `http://www.rsasecurity.com/rsalabs/challenges/factoring/`

[21] Robert D. Silverman, *A cost-based security analysis of symmetric and asymmetric key lengths*, Bulletin 13, RSA Security, 2000, `http://www.rsasecurity.com/rsalabs/bulletins/bulletin13.html`