Factoring Large Numbers with the TWIRL Device (preliminary draft)

Adi Shamir, Eran Tromer

Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot 76100, Israel
{shamir,tromer}@wisdom.weizmann.ac.il

January 23, 2003

Abstract. The security of the RSA cryptosystem depends on the difficulty of factoring large integers. The best current factoring algorithm is the Number Field Sieve (NFS), and its most difficult part is the sieving step. In 1999 a large distributed computation involving thousands of workstations working for many months managed to factor a 512-bit RSA key, but 1024-bit keys were believed to be safe for the next 15-20 years. In this paper we describe a new hardware implementation of the NFS sieving step (based on standard 0.13μm, 1 GHz VLSI technology) which is 3-4 orders of magnitude more cost effective than the best previously published designs (such as the optoelectronic TWINKLE of [13] and the mesh-based sieving of [5]). Based on a detailed analysis of all the critical components (but without an actual implementation), we believe that the NFS sieving step for 1024-bit RSA keys can be completed in less than a year by a $10M device, and that the NFS sieving step for 512-bit RSA keys can be completed in less than ten minutes by a $10K device. Coupled with recent results about the difficulty of the NFS matrix step [10], this raises some concerns about the security of these key sizes.

1 Introduction

The hardness of integer factorization is a central cryptographic assumption and forms the basis of several widely deployed cryptosystems. The best known integer factorization algorithm is the Number Field Sieve [8], which was successfully used to factor a 512-bit RSA modulus [3]. However, it appears that a PC-based implementation of the NFS cannot practically scale much further, and specifically its cost for 1024-bit composites is prohibitive. Recently, the prospect of using custom hardware for the computationally expensive steps of the Number Field Sieve has gained much attention. While mesh-based circuits for the matrix step have rendered that step quite feasible for 1024-bit composites [1, 10], the situation is less clear concerning the sieving step. While several sieving devices have been proposed, including TWINKLE [13, 9] and a mesh-based circuit [5], apparently none of these can practically handle 1024-bit composites.

One lesson learned from Bernstein's mesh-based circuit for the matrix step [1] is that it's 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. We propose a new device that combines this intuition with the TWINKLE-like approach of exchanging time and space.

Whereas TWINKLE tests one sieve location serially, the new device handles thousands of sieve locations in parallel at every clock cycle. In addition, it is smaller and easier to construct: for 1024-bit composites we can fit 44 independent sieving devices on a 30cm single silion wafer, whereas each TWINKLE devices requires a full GaAs wafer even for 512-bit composites. While our approach is related to [5], it scales better and avoids some thorny issues.

The main difficulty is how to use a single copy of the input (or a small number of copies) to solve many subproblems in parallel, without collisions or long propagation delays and while maintaining storage efficiency. We address this with a heterogeneous design that uses a variety of routing circuits and takes advantage of available technological tradeoffs. The resulting cost estimates suggest that for 1024-bit composites the sieving step may be surprisingly feasible.

Section 2 reviews the sieving problem and the TWINKLE device. Section 3 describes the new device, called TWIRL[1], and Section 4 gives some preliminary cost estimates. Appendix A discusses some additional issues in the operation of TWIRL. Appendix B specifies the assumptions used for the cost estimates. Appendix C sketches some suboptimal yet interesting alternative designs, and Appendix D relates this work to previous ones.

2 Context

2.1 Sieving in the Number Field Sieve

Our proposed device implements the sieving substep of the NFS relation collection step, which in practice is the most expensive part of the NFS algorithm [10]. We begin by reviewing the sieving problem, in a greatly simplified form and after appropriate reductions.[2] See [8] for background on the Number Field Sieve.

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 and 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 $a \in \{0,\dots,R-1\}$ that are members of many progressions P_i with large p_i:

$$g(a) > T \;\text{ where }\; g(a) = \sum_{i:a\in P_i} \log p_i$$

It is permissible to have "small" errors in this threshold check.

In the NFS relation collection step we need to solve $2H$ instances of the sieving problem, divided into H pairs of instances. Each pair consists of a *rational sieve* and an *algebraic sieve*, both of the above form but with different progressions, B and T. For each pair of sieves, each value a that passes the threshold in both sieves implies a *candidate*. Each candidate undergoes additional tests, for which

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

[2] The description matches both line sieving and lattice sieving. However, for lattice sieving we may wish to take a slightly different approach (cf. A.4).

it is beneficial to also know the set $\{i : a \in P_i\}$ (for each sieve separately). 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.

One of the parameters in the NFS algorithm is the number of "large primes" allowed. In our context, the effect is as follows: increasing the number of allowed "large primes" decreases the work of sieving per se (as expressed by the parameters R,B,H above), but increases the frequency of candidates, decreases the fraction of candidates that yield relations and makes the testing of each candidate harder. In particular, for NFS with k "large prime per side", $k > 1$, the tests of candidates involve *cofactor factorization*: factoring integers somewhat larger than B^k.

2.2 TWINKLE

Since TWIRL follows the TWINKLE approach of exchanging time and space compared to traditional NFS implementations, it will be convenient to review it first.

The TWINKLE device operates essentially as follows, again after considerable simplification. The device consists of a wafer containing numerous independent cells, each in charge of a single progression P_i. After initialization the device operates 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, 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; clocking and feedback are done using digital optics; the rest is implemented by digital electronics. To support the optoelectronic operations, TWINKLE uses Gallium Arsenide wafers which are small, expensive and hard to manufacture compared to silicon wafers, which are readily available.

3 The New device

3.1 Approach

We will now describe the TWIRL device. 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 Section 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.

We want to solve $2H \langle\!\langle \approx 1.1 \cdot 10^{10} \rangle\!\rangle$ instances of the sieving problem with smoothness bound $B \langle\!\langle = 2.2 \cdot 10^8 \rangle\!\rangle$ and sieving line width $R \langle\!\langle = 1.1 \cdot 10^{14} \rangle\!\rangle$. Consider first a device that handles one sieve location per clock cycle, like TWINKLE, but does so using a pipelined systolic chain of electronic adders. Such a device would consist of a long unidirectional bus, $\log T \langle\!\langle = 6 \rangle\!\rangle$ 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.

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

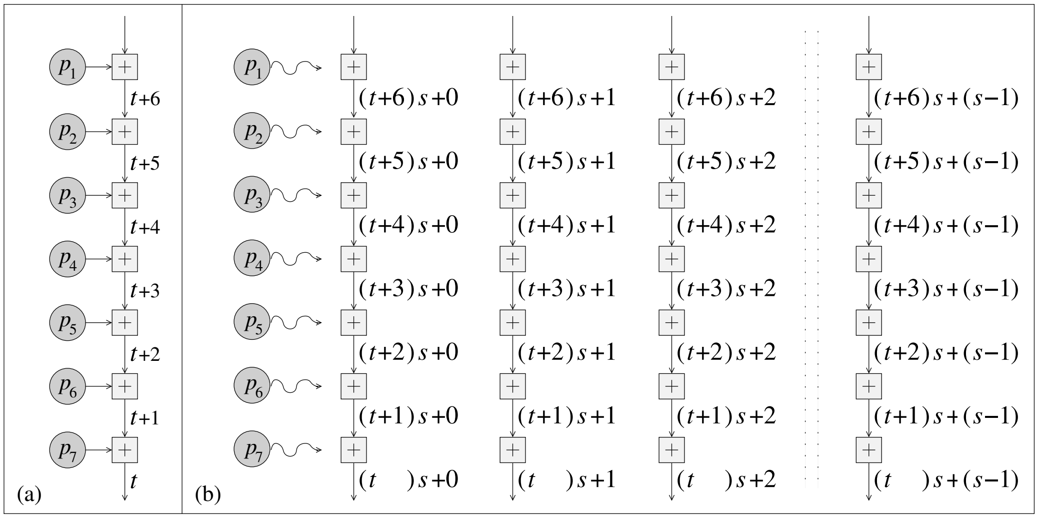

Fig. 1. Flow of sieve locations through the device in (a) a chain of adders and (b) TWIRL.

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 $g(1), \dots, g(R)$, one value per clock cycle.[4]

We reduce the run time by a factor of s $\langle\!\langle = 1024\rangle\!\rangle$ by handling the sieving range $0, \dots, R-1$ in chunks of length s, as follows. The bus is thickened by a factor of s to contain s logical lines of $\log T$ 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 $(t-z)s + i$, $i \in \{0, \dots, s-1\}$. The first values to appear at the end of the pipeline are $\{g(0), \dots, g(s-1)\}$; they appear simultaneously, followed by successive disjoint groups of size s, one group per clock cycle. See Fig. **??**.

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. 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 (see Fig. 1). Using standard VLSI technology, we can get about $\langle\!\langle 44\rangle\!\rangle$ independent sieving devices from each 30cm silicon wafer (whose manufacturing cost is about \$5,000), though some of these devices may be defective (cf. A.5).

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

[4] This variant of TWINKLE was considered in [9], but deemed inferior in that context.

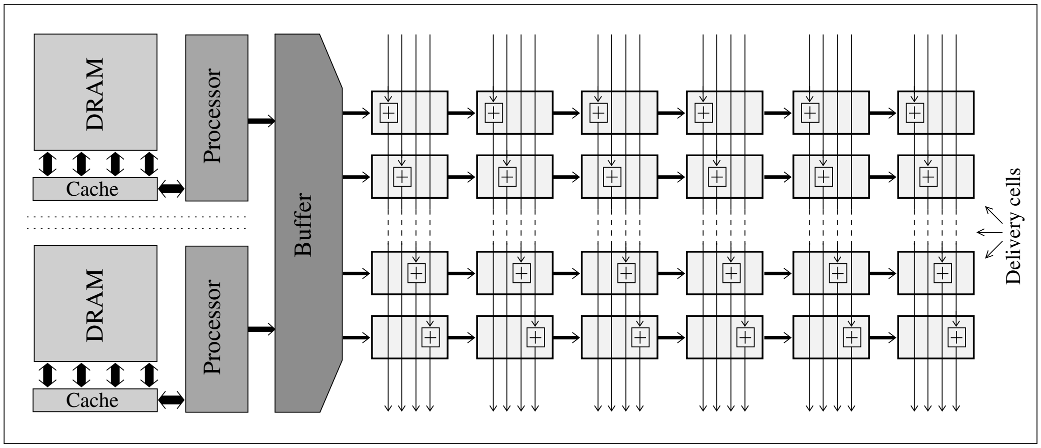

Fig. 2. Schematic structure of a largish station.

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*.[5]

The following subsections 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 Appendix A.

3.2 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 $\langle\!\langle p_i > 128s \rangle\!\rangle$, it is beneficial to use expensive logic circuitry that handles many progressions but allows very compact storage of each progression. The resultant architecture is shown in Fig. 2. Each progression is represented as a *progression triplet* that is stored in a memory bank, using compact DRAM 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 are passed into *buffers* that merge the outputs of several processors, perform fine-tuning of the timing and create *delivery pairs.* 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.

Scanning the progressions. The progressions are partitioned into many $\langle\!\langle 2{,}000 \rangle\!\rangle$ DRAM banks, where each bank contains some d progression $\langle\!\langle 32 \leq d < 53{,}204 \rangle\!\rangle$. A progression P_i is represented by a progression triplet of the form (p_i, ℓ_i, τ_i), where

[5] These are not to be confused with the "large" and "small" primes of the high-level NFS algorithm — all the primes with which we are concerned here are "small" (rather than "large" or in the range of "special-q").

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 the value $\ell_i = a_i \bmod s$ is the number of the corresponding bus line

The processor repeats the following operations, in a pipelined manner:

1. Read and erase a 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 this processor.
3. Compute $\ell' \leftarrow (\ell + p) \bmod s$ and $\tau_i' \leftarrow \tau_i + \lfloor p/s \rfloor + w$, where $w = 1$ is 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 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's 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 explained below.

Progression storage. The processor reads progression triplets from the memory in sequential cyclic order and at a constant rate ⟪of one triplet every 2 clock cycles⟫. After its new values are calculated, a progression is stored at a different memory location — 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 location within a short distance of any location, we increase the amount of memory ⟪by a factor of 2⟫; the progression is stored at the first unoccupied location, starting at the one that will be read at time τ_i' and going backwards cyclically. When the processor reads an empty memory location, it does nothing else at that iteration.

In simulations for primes close to B ⟪$= 2.2 \cdot 10^8$⟫ (resp., ⟪200,000⟫), the distance between the first unoccupied location and the ideal location had average ⟪0.50⟫ (resp., ⟪0.71⟫), its standard deviation was ⟪1.34⟫ (resp., ⟪1.33⟫), and it was smaller than 32 (resp., ⟪16⟫) for all but ⟪0.02%⟫ (resp., ⟪0.007%⟫) of the iterations. Note that in the sieving problem it is permissible to miss some of the contributions.

It would be inefficient to have the processor test several addresses near its target until it finds an unoccupied one. We would like the processor to just output the ideal target address, and then some local autonomous circuitry inside the memory would route the value to the first unoccupied location. To do so without a large overhead for every memory location, and to maintain a high clock rate, 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, since it is useless to make it larger (progression triplets are written at most $p_{\max}/s$ memory locations ahead of the current read location, in cyclic order). With this choice, triplets with $p_i = p_{\max}$ are stored right before the current read location; 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 $\ln(p_{\max}) \leq \ln(B)$ 《≈ 19.2》 is much smaller than s 《$= 1024$》, memory access always occurs at a small window that slides at a constant rate of one memory location every 《2》 clock cycles.

This sliding window is handled by a fast SRAM-based cache. 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 achieves a very high DRAM bandwidth and hides the high DRAM latency.[6] Note that cache misses cannot occur. The only interface between the processor and memory are the operations "read next memory location" and "write triplet to first unoccupied memory location before the given address". The logic for the latter is implemented within the cache, using auxiliary per-triplet occupancy flags and some local pipelined circuitry.

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. If there are multiple emission triplets with identical ℓ and τ values, they may be merged by summing their $\lfloor \log p_i \rceil$ values.

Buffer units can be be realized as follows. First, all incoming emission triplets are placed in a parallelized priority queue indexed by τ, implemented as a small mesh whose rows are continuously bubble-sorted and whose columns undergo random local shuffles. The elements in the last few 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 appropriate choices of parameters this should happen seldom.

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》.

[6] For the stations that handle the smaller primes in the "largish" range, it might be beneficial to increase the cache size to d and eliminate the DRAM.

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.

Most of the time the cells in a delivery line act as shift registers, and their adders are unutilized. Thus, we can reduce the cost of adders and registers by interleaving. We use larger delivery cells that span $r \langle\!\langle = 4 \rangle\!\rangle$ adjacent bus lines, and contain an adder just for the q-th line among these, with q fixed throughput the delivery line and incremented cyclically in the subsequent delivery lines. The buffer is entrusted with the role of sending delivery pairs to delivery lines that have an adder at the appropriate bus line. As a bonus, we now put every r adjacent delivery lines in a single bus pipeline stage, so that it contains one adder per bus line. This reducing 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[7] 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.

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.

Adders. Logically, each bus line carries a $\log T \langle\!\langle = 6 \rangle\!\rangle$-bit integer. These are encoded by a redundant representation, as a pair of $\log T$-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 extra bus wires due to the redundant representation can be accommodated using multiple metal layers of the silicon wafer.[8]

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

[7] After the change made in Appendix A.1 this becomes $\lfloor \mathrm{rev}(a \bmod s)/r \rfloor$, where $\mathrm{rev}(\cdot)$ denotes bit-reversal of $\log_2 s$-bit numbers and s,r are powers of 2.

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

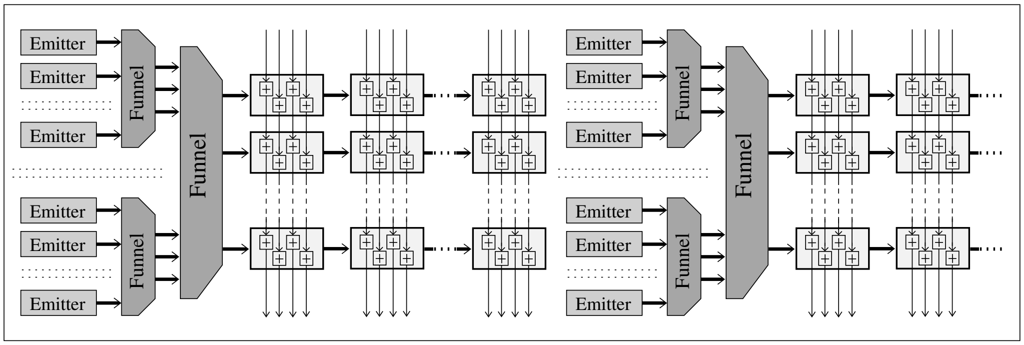

Fig. 3. Schematic structure of a smallish station.

become 1 (in which case the sum is at least T), further additions do not switch them back to 0.

Notes. For our choice of parameters, the area occupied by the DRAM banks, processors and buffers is ⟪2.3 times⟫ larger than the area occupied by delivery lines. Thus, the delivery lines cannot be packed close together. To avoid wasting chip area, we can physically place the processors, buffers and DRAM banks underneath the bus wires, using the multiple layers available on silicon wafers.

In the description of the processors, DRAM and buffers, we took the τ values to be arbitrary integers designating clock cycles. Actually, it suffices to maintain these values modulo some integer ⟪2048⟫ 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 $\log_2 p_i +$ ⟪$\log_2 2048$⟫ bits of DRAM for the progression triplet, plus $\log_2 p_i$ bits for re-initialization (cf. A.2).

The amortized circuit area per largish progression is $\Theta(s^2(\log s)/p_i + \log s + \log p_i)$.[9] For fixed s this equals $\Theta(1/p_i + \log p_i)$, and indeed for large composites the overwhelming majority of progressions ⟪99.2%⟫ will be handled in this manner.

3.3 Smallish Primes

For progressions with p_i close to s, ⟪$256 < p_i < 128s$⟫, each proessor can handle very few progressions because it can produce at most one emission triplet every ⟪2⟫ clock cycles. Thus, the amortized cost of the processor, memory control circuitry and buffers is very high. Moreover, such progression cause emissions so often that communicating their 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 another station design, which differs in several ways from the largish stations (see Fig 3).[10]

Emitters. 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*,

[9] 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.

[10] See Appendix C for alternatives.

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 occasionally emits delivery pairs of the form $(\lfloor \log p_i \rceil, \ell)$ which indicate that the value $\lfloor \log p_i \rceil$ should be added to the ℓ-th bus line some fixed time interval later. Appendix A.1 describes a compact emitters design.

Funnels. Each emitter is continuously updating its internal counters, but it creates a delivery pair only once per roughly $\sqrt{p_i}$ 《between 8 and 128》 clock cycles (see below). 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.

We use the following implementation. 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. 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 if this would not overwrite another value. The t-th output array is read off the rightmost column at time $t+n$.

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 probability ν independently of the others (as discussed below, $\nu \approx 1/\sqrt{p_i}$), 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 《$m = 5$》 rows and 《$n \approx 1/\nu$》 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.

The above funnels have a suboptimal compression ratio n/m 《$\approx 1/5\nu$》, i.e., the probability ν' 《$\approx 1/5$》 of a funnel output value being non-empty is still rather low. We thus feed these output into a second-level funnel 《with $m' = 35$, $n' = 14$》, 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 by a factor of $5 \cdot 14/34 = 2$, so the number of delivery lines is twice the naive optimum. We do not interleave the adders in the delivery lines as done for largish stations, in order to avoid the overhead of directing delivery pairs to an appropriate delivery line.[11]

[11] 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.

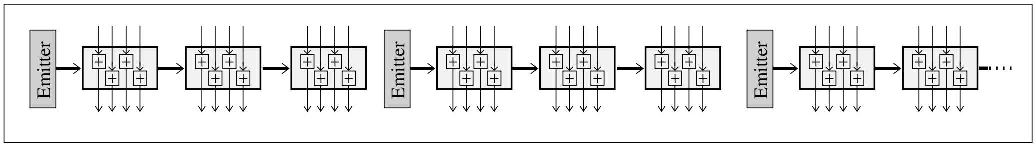

Fig. 4. Schematic structure of a tiny station, for a single progression.

Duplication. The other major change is 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 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, there is a corresponding decrease in the emission frequency of any specific emitter, which allows us to handle p_i close to (or even smaller than) s.

The asymptotic cost of a 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 large p_i.

3.4 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. This leads to a third station design for the tiny primes $\langle\!\langle p_i < 256 \rangle\!\rangle$. While there are few such progressions, their contributions are significant due to their very small periods.

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 4). These emitters are a degenerate form of the ones used for smallish progressions (cf. A.1). 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 cost is dominated by the delivery lines, it is $\Theta(s)$ regardless of p_i.

This concludes the description of the station types. There are some additional considerations that require minor (though important) adaptations of the design, and these are discussed in Appendix A.

[12] $\langle\!\langle n_i = s/2\sqrt{p_i} \rangle\!\rangle$ rounded to a power of 2 (cf. A.1), which is in the range $\langle\!\langle \{1,\ldots,32\} \rangle\!\rangle$.

[13] The segment length is the largest power of 2 smaller than p_i (cf. A.1).

4 Cost estimates

Having outlined the design and specified the problem size, we next estimate the cost of a hypothetical TWIRL device using today's VLSI technology. The hardware parameters used are specified in Appendix B.1. 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.

4.1 Cost of Sieving for 1024-bit Composites

We use the following NFS parameters: $B = 2.2 \cdot 10^8$, $R = 1.1 \cdot 10^{14}$, $2H \approx 1.1 \cdot 10^{10}$. This choice is explained in Appendix B.2.

With these parameters and $s = 1024$, one TWIRL device requires 1,423mm^2 of silicon wafer area. Of this, 71% is occupied by the largish progressions (and specifically, 21% of the device is used for the DRAM banks), 24% is used by the smallish progressions and the rest (5%) is used by the tiny progressions. Various parameters of this design are mentioned throughout Section 3.

A single TWIRL device would require 110 seconds per sieve line, for a total of 40,000 years for the full sieving. Since 44 full devices can be manufactured on a single 30cm silicon wafer at a cost of about \$5,000, for \$4.5M we can build 40,000 independent devices that, when run in parallel, would complete the sieving task within 1 year.

After including the initial costs of design and production, accounting for the power supply and cooling systems, adding the cost of PCs for collecting the data and leaving a generous error margin,[14] it appears realistic that all the sieving required for factoring 1024-bit integers can be completed within 1 year at a total cost of less than \$10M.

We also consider a TWIRL device for NFS without large primes and with an increased smoothness bound of $B = 10^9$ (which is an overestimate; cf. B.2). This simplifies the post-processing of candidates. Here, cost is dominated by DRAM so we increase s to 4096. It could be beneficial to increase it further, but here the width of the bus already approaches the diameter of a 30cm wafer. For these parameters a single TWIRL devices occupies half a wafer, 52% of which are occupied by DRAM. By running 40,000 such devices (using \$26M worth of wafers) in parallel, we can complete the sieving within 1 year. After doubling the cost to account for overheads, as above, we get an estimate of \$50M.

4.2 Implications for 1024-bit Composites

It has been often claimed that 1024-bit RSA keys are safe for the next 15 to 20 years, since both NFS relation collection and the NFS matrix step would remain unfeasible (e.g., in [15] and in a NIST guideline draft [12]). Our evaluation suggests

[14] It is a common rule of thumb that the total cost is twice the silicon cost.

that sieving can be achieved within one year at a total cost of \$10M, and previous work [10] suggests that the matrix step is easier — 6 hours using a single-wafer device that costs \$5,000, for the same NFS parameters. While these estimates are hypothetical and rely on ferocious extrapolations, they should be taken into account by anyone planning to use a 1024-bit RSA key.

With efficient custom hardware for both sieving and the matrix step, other subtasks in the NFS algorithm may emerge as bottlenecks. One such subtask is the cofactor factorization of the candidate relations identified during sieving (cf. 2.1). In TWINKLE with line sieving [9] this was indeed the bottleneck — many PCs working in parallel were needed to keep up with the rate at which TWINKLE emitted candidate relations. Since TWIRL is faster by orders of magnitude, this is potentially a significant obstacle. Quite possibly, the cost of cofactor factorization could be reduced by algorithmic means [2] or by using custom hardware; this certainly deserves further investigation. Another potentially expensive step is the merging of partial relations. Note, however, that both of these steps are needed only when the "large primes" variant of NFS is used. These potential problems can thus be completely eliminated by using a more expensive (\$50M) TWIRL device which performs sieving for NFS without large primes; this cost is not unfeasible either.

There is considerable need for more concrete estimates of the 1024-bit NFS sieving parameters, and the only way to learn the precise costs involved would be to perform a factorization experiment. None the less, it currently appears difficult to identify any specific issue that may prevent a sufficiently motivated and well-funded party from applying the Number Field Sieve to 1024-bit composites.

4.3 Cost of Sieving for 512-bits Composites

Since several hardware designs [13, 9, 7, 5] were proposed for the sieving of 512-bit composites, it would be instructive to obtain cost estimates for TWIRL with the same problem parameters.

We assume the same parameters as in [9]: $B = 2^{24} \approx 1.7 \cdot 10^7$, $R = 1.8 \cdot 10^{10}, 2H = 1.8 \cdot 10^6$. We set $s = 1024$ and use the same cost estimation expressions that lead to the figures quoted for 1024-bit composites.

A single TWIRL device would have a die size of about 849mm^2, 58% 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 9 hours.

For these NFS parameters TWINKLE would require 1.8 seconds per sieve line, the FPGA-based design of [7] would require about 10 seconds and the mesh-based design of [5] would require 0.36 seconds. To provide a fair comparison to TWINKLE and [5], we should consider a single wafer full of TWIRL devices running in parallel. Since we can fit 74 of them, the effective time per sieve line is reduced to 0.00023 seconds.

Thus, in factoring 512-bit composites TWIRL is about 1600 times more cost effective than the best previously published design [5], and 7800 times more cost effective than TWINKLE. This is discussed further in Appendix D.

4.4 Cost of Sieving for 768-bits Composites

To obtain the sieving parameters for 768-bit composites, we extrapolate from [9] as done for 1024-bit integers[15]. We get $B = 1.2 \cdot 10^7$, $R = 2.9 \cdot 10^{12}$ and $2.9 \cdot 10^8$ sieving runs.

We choose a smaller $s = 256$ to obtain a device with die size of 154mm^2 whose cost is dominated by the largish progressions. We can fit 411 devices per wafer, where each device independently handles a sieve line in 11.5 seconds. Thus, the device from a single wafer can complete the sieving task within 95 days. This would cost \$5,000, which incidentally is one tenth of the RSA-768 challenge prize [14].[16]

4.5 Larger composites

For largish progressions, the amortized circuit area per progression is $\Theta(s^2(\log s)/p_i + \log s + \log p_i)$ with small constants (cf. 3.2). For smallish progressions it is $\Theta(\,(s/\sqrt{p_i} + 1)\,(\log s + \log p_i)\,)$ with much larger constants (cf. 3.3). 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 $s = \Theta(\sqrt{B})$ to get a speed advantage of $\Theta(\sqrt{B})$ over serial implementations, while maintaining the small constants. Indeed, we can keep increasing s essentially for free until the area of the largish processors, buffers and delivery lines becomes comparable to the area occupied by the DRAM that holds the progression triplets.

For some range of input sizes, it may be 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 applicable NFS polynomial).

As a practical issue, at some point the device would exceed the capacity of a single silicon wafer. 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.

5 Conclusion

We have presented a new design for a custom-built sieving device. The device consists of a thick pipeline that carries sieve locations through thrilling adventures, where they experience the addition of progression contributions in myriad different ways that are optimized for various scales of progression periods. In factoring 512-bit integers, the new device is 1600 times faster than best previously published designs. For 1024-bit composites and appropriate choice of NFS parameters, the new device can complete the sieving task within 1 year at a cost of \$10M, thereby raising some concerns about the security of 1024-bit RSA keys.

[15] Including the transition to the "small matrix" NFS parameters choice. For standard NFS parameters and $s = 1024$ we get similar total area and run times, but there it is obtained by a few large TWIRL devices rather than many small ones.

[16] Needless to say, this disregards an initial cost of over \$1M.

Acknowledgments. 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. We are indebted to Arjen K. Lenstra for many conversations about NFS.

References

1. Daniel J. Bernstein, *Circuits for integer factorization: a proposal*, manuscript, 2001, `http://cr.yp.to/papers.html`
2. Daniel J. Bernstein, *How to find small factors of integers*, manuscript, 2000, `http://cr.yp.to/papers.html`
3. 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*, proceedings of Eurocrypt 2000, LNCS **1807** 1–17, Springer-Verlag, 2000
4. Don Coppersmith, *Modifications to the number field sieve*, Journal of Cryptology **6** 169–180, 1993
5. Willi Geiselmann, Rainer Steinwandt, *A dedicated sieving hardware*, proceedings of PKC 2003, LNCS **2567** 254–266, Springer-Verlag, 2002
6. International Technology Roadmap for Semiconductors 2001, `http://public.itrs.net/`
7. Hea Joung Kim, William H. Magione-Smith, *Factoring large numbers with programmable hardware*, proceedings of FPGA 2000, ACM, 2000
8. Arjen K. Lenstra, H.W. Lenstra, Jr., (eds.), *The development of the number field sieve*, Lecture Notes in Math. **1554**, Springer-Verlag, 1993
9. Arjen K. Lenstra, Adi Shamir, *Analysis and Optimization of the TWINKLE Factoring Device*, proceedings of Eurocrypt 2002, LNCS **1807** 35–52, Springer-Verlag, 2000
10. Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, Eran Tromer, *Analysis of Bernstein's factorization circuit*, proceedings of Asiacrypt 2002, LNCS **2501** 1–26, Springer-Verlag, 2002
11. Brian Murphy, *Polynomial selection for the number field sieve integer factorization algorithm*, Ph. D. thesis, Australian National University, 1999
12. National Institute of Standards and Technology, *Key management guidelines, Part 1: General guidance (second draft)*, June 2002, `http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html`
13. Adi Shamir, *Factoring large numbers with the TWINKLE device (extended abstract)*, proceedings of CHES'99, LNCS **1717** 2–12, Springer-Verlag, 1999
14. RSA Security, *The new RSA factoring challenge*, web page, Jan. 2003, `http://www.rsasecurity.com/rsalabs/challenges/factoring/`
15. Robert D. Silverman, *A cost-based security analysis of symmetric and asymmetric key lengths*, Bulletin 13, RSA laboratories, 2000, `http://www.rsasecurity.com/rsalabs/bulletins/bulletin13.html`

A Additional Design Considerations

A.1 Implemention of Emitters

The designs of smallish and tiny progressions (cf. 3.3, 3.4) 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 subsection describes our proposed emitter implementation. For context, we first describe some less efficient designs.

Straightforwards implementations. One simple implementaion would be to keep a $\lceil \log_2 p_i \rceil$-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 [13], 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 is irregular, and would require expensive circuitry to compute.

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 $\alpha \langle\!\langle = 10 \rangle\!\rangle$ and β_i. The residues modulo s are assigned to bus lines with bit-reversed indices; that is, sieve locations congruent modulo s to w are handled by the bus line at physical location $\text{rev}(w)$, where

$$w = \sum_{i=0}^{\alpha-1} c_i 2^i \ , \quad \text{rev}(w) = \sum_{i=0}^{\alpha-1} c_{\alpha-1-i} 2^i \quad \text{for some } c_0, \ldots, c_{\alpha-1} \in \{0,1\}$$

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. The advantage of this choice is the following.

Lemma 1. *For any fixed progression with $p_i > 2$, the emissions destined to any fixed group occur at regular time intervals of $T_i = \lfloor 2^{-\beta_i} p_i \rfloor$, up to an occasional delay of one clock cycle due to modulo s effects.*

Proof. Emissions into the j-th group correspond to sieve locations $a \in P_i$ that fulfill $\lfloor \text{rev}(a \bmod s)/2^{\beta_i} \rfloor = j$, which is equivalent to $a \equiv c_j \pmod{2^{\alpha-\beta_i}}$ for some c_j. Since $a \in P_i$ means $a \equiv r_i \pmod{p_i}$ and p_i is coprime to $2^{\alpha-\beta_i}$, by the Chinese Remainder Theorem we get that the set of such sieve locations is exactly $P_{i,j} \equiv \{a : a \equiv c_{i,j} \pmod{2^{\alpha-\beta_i} p_i}\}$ 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 $\Delta = \lfloor a_2/s \rfloor - \lfloor a_1/s \rfloor$. If $(a_2 \bmod s) > (a_1 \bmod s)$ then $\Delta = \lfloor (a_2 - a_1)/s \rfloor = \lfloor 2^{\alpha-\beta_i} p_i / s \rfloor = T_i$. Otherwise, $\Delta = \lceil (a_2 - a_1)/s \rceil = T_i + 1$. $\square$

Note that $T_i \approx \sqrt{p_i}$, by the choice of β_i.

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

- Counter A operates modulo $T_i = \lfloor 2^{-\beta_i} p_i \rfloor$ ⟪typically 7 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 9 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.

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[17]. The combined length of the counters is roughly $\log_2 p_i$ bits, and with appropriate adjustments they can be implemented using compact ripple adders[18] as in [9].

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 $T_i = 1$, i.e., an emission occurs every one or two clock cycles. The emitter circuitry is identical to the above; note that Counter A has become zero-sized (i.e., a wire), which leaves a single counter of size $\beta_i \approx \log_2 p_i$ bits.

A.2 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 [13] this became a bottleneck). We can avoid this by noting, as in [5], 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 $\log_2 p_i$ bits per progression (this is included in our cost estimates) and the addition

[17] For sieving the rational side of NFS, it suffices to fix the smoothness bounds. Similarly for the preprocessing stage of Coppersmith's Factorization Factory [4] .

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

can be done efficiently using on-wafer special-purpose processors. Since the interval R/s beetween 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 (cf. A.4).

A.3 Processing Candidates

Having computed approximations of the sum of logarithms $g(a)$ 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 (cf. 2.1). These are implemented as follows.

Identifying candidates. 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 $g(a) > T$. We may expect several of these per clock cycle. Recall that instances of the sieving problem occur in pairs ("rational sieve" and "algebraic sieve"), and candidates correspond to a values that pass the threshold on both sieves. Accordingly, we always operate TWIRL devices in pairs, with synchronized clocks. 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. On average, we may expect at most one candidate per several dozen clock cycles (considerably more seldom for NFS with zero or one "large primes" per side, cf. 2.1).

Finding the corresponding progressions. For each candidate we need to compute the set $\{i : a \in P_i\}$, separately for the rational and algerbaic 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.[19] It thus appears 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 $\log_2 s$ bits, goes upstream (i.e., opposite to the flow of values in the bus) from the threshold comparators . The *divisors channel*, of width $\log_2 B$ 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 $\log_2 B$-bit values. Every clock cycle, the processor writes a value to its diary: if the processor inserted an emission triplet $(\lfloor \log p_i \rceil, \ell_i, \sigma_i)$ into the buffer at this clock cycle, it writes the pair (p_i, ℓ_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 reads the diary value z places before the last value written (in cyclic order), where z is the latency (in pipeline stages) from the processor's output into the buffer, through the bus and threshold comparators and back to the processor

[19] Namely, by finding the small factors of $F_j(a - R, b)$ where F_j is the relevant NFS polynomial and b is the line being sieved.

through the lines channel. If the read diary entry is a non-NULL value (p_i, ℓ_i) and $\ell_i = \ell$ then the processor transmits p_i downstream via the divisors channel.

Whenever candidates occur at time intervals smaller than the sum of the latencies of the two channels $\langle\!\langle 2 \cdot 2000 \rangle\!\rangle$, the corresponding contributions into the divisors channel will be intermingled (as seen at the device output), and collisions may occur. Non-destructive mixing is not a problem, since by appropriate assignment of primes to largish processors we can reconstruct the exact timing from the p_i values. If collisions occur too often, we can split each of the two channels into several disjoint pipelines, each passing through a small portion of the primes.

The maximum diary size (if the above splitting is not done) is about $\langle\!\langle 2640 \rangle\!\rangle$ entries. The diaries are implemented as DRAM banks of a degenerate form: they perform row enumeration autonomously using a pair of "tokens", one for read and one for write, that moves along the rows at a constant rate. The only interface, apart from initialization and data I/O, is an instruction whether to perform a read operation or a write operation. Whenever a read occurs a diary value update is missed (through the write token does progress); the effect is negligible. To keep up with the throughput of one value per clock cycle, we need to interleave the diary entries across several DRAM banks. However, we can share the row control circuitry of DRAM banks among several adjacent processors. In any case, the diaries and reporting channels occupy a small fraction of the total device area.

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" (cf. 2.1). With the exception of cofactor factorization, these tasks can be effectively handled by special-purpose processors and pipelines, which are similar to the division pipeline of [5, Section 4] except that here we have far fewer candidates (cf. D.4). Cofactor factorization, if needed, may become a significant bottleneck; fortunately it can be avoided at a reasonable cost (cf. 4.2). This aspect deserves further exploration.

A.4 Lattice Sieving

The above is motivated by NFS line sieving, which has very large sieve line length R. Lattice sieving (i.e., "special-q") involves fewer sieving locations. Moreover, it makes cofactor factorization easier (cf. 4.2). However, lattice sieving has very short sieving lines (8192 in [3]), so the natural mapping to the lattice problem as defined here (i.e., lattice sieving by lines) leads to values of R 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 is imminent.

Re-initialization is needed only when the special-q lattices are changed (every $8192 \cdot 5000$ sieve locations in [3]), but 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.

A.5 Fault Tolerance

For large VLSI designs, we should expect the presence of local defects and include means of handling them. For our choice of 1024-bit parameters a single silicon wafer contains about ⟪44⟫ independent sieving devices, which can be separated and operated individually. Given the low fault density of current technology, most of these devices will be usable even if some parts of the circuit are devoid of contingency plans.

To increase the yield of good devices, we make the following adaptations. 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. For small devices we can expect few faults, and thus we may ignore the outputs of faulty bus lines. For larger devices 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 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.

B Parameters for Cost Estimates

B.1 Hardware

The hardware parameters used are those given in [10] (which are consistent with [6]): standard 30cm silicon wafers with 0.13μm 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 $0.2\mu\text{m}^2$ per bit (averaged over the area of DRAM banks). For 512-bit composites we will use logic-type wafers, with transistor density of $2.38\mu\text{m}^2$ per transistor and DRAM density of $0.7\mu\text{m}^2$ per bit. The clock rate is 1GHz clock rate, which appears realistic with judicious pipelining of the processors.

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 Section 3. A typical largish processor is assumed to require 58,000 transistors (including DRAM column logic, amortized buffer area and the amount of cache memory that are independent of p_i). A typical emitter is assumed to require 1,650 transistors in a smallish station (including the amortized costs of funnels), and 462 in a tiny station. Delivery cells are assumed to require 388 transistors with interleaving (i.e., in largish stations) and 800 without interleaving (i.e., in smallish and tiny stations). We assume that the memory system of Section 3.2 requires 2.5 times more area per useful bit than standard DRAM, due to the required slack and 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 0.5 wires per μm, possibly achieved by using multiple metal layers.

Note that 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 types of units is very small compared to designs that are commonly handled by the VLSI industry, and there is considerable room for variations.

B.2 Sieving Parameters for 1024-bit Composites

Our estimate of the sieving parameters for 1024-bit composites are as follows. The smoothness bound is $B = 2.2 \cdot 10^8$. The sieve line width is $R = 1.1 \cdot 10^{14}$. The number of sieving instances (i.e., twice the number of sieve lines) is $2H = 1.1 \cdot 10^{10}$. These are derived as follows.

We start with the estimated parameters of 512-bit line sieving from [9], as cited in Section 4.3. These parameters appear to be quite conservative compared to actual experiments [3]. They use 2 large primes on the rational side and 3 on the algebraic side. Extrapolation is then performed similarly to the "small matrix" of [10, Section 5.1, 5.2]:

First, to account for the larger problem size we increase the smoothness bound B, the length of sieving lines R and the number of sieve lines by a factor of $L_{2^{1024}}[1/3, 2/3^{2/3}]/L_{2^{512}}[1/3, 2/3^{2/3}] \approx 2737$. We then adjust to a different choice of NFS parameters, corresponding to optimized throughput cost with serial sieving and mesh-based matrix step. This decreases the smoothness bound B by a factor[20] of $L_{2^{1024}}[1/3, (5/3)^{1/3}(2/3)]/L_{2^{1024}}[1/3, 2/3^{2/3}] \approx 210$, but increases both the sieve line width R and the number of sieve lines H by a factor of $L_{2^{1024}}[1/3, (5/3)^{1/3}(5/6)]/L_{2^{1024}}[1/3, 4/3^{2/3}] \approx 2.3$. For most of the primes, we assume that there is one progression per prime smaller than B (this holds on average). For very small primes

[20] In [10] a decrease by an additional factor of 2.2 was made, to account for the observed behavior of the $o(1)$ in the NFS exponents. Also, in [10] the extrapolation was applied to the number of primes rather than to the smoothness bounds; that would lead to B that is smaller by another factor of 1.4 (due to the density of primes). To remain on the safe side, we forego both factors.

this is false due to the NFS polynomial selection method [11], so we use the distribution of roots from the 512-bit factorization experiment [3]. We also add a few progressions for powers of small primes.

Note that while our extrapolation to the "small matrix" parameters is legitimate, it is not clear why it should be beneficial. Using the notation of [10], in our case the relation collection cost is $L(2\alpha + \beta/2)$ (cf. 4.5), and we would thus expect it to be minimized by the standard NFS parameter choice. Still, using the "small matrix" NFS parameters halves the cost. Moreover, the standard parameters lead to a full-wafer device while the "small matrix" leads to much smaller devices (i.e., each wafer can be cut into many separate dies) — this is much better in terms of cooling and fault tolerance. Also, the matrix step becomes easier.

We consider a second case, where the NFS algorithm is used without the "large primes" variant (not to be confused with the "largish primes" of TWIRL, which are smaller). Here we simply take the smoothness bound of 512-bit factorization to be 10^9, which is the large prime bound used in [3]; this is certainly a gross overestimate. We then extrapolate twice as before, to get $B = 1.3 \cdot 10^{10}$.

C Alternative designs

We have considered several alternatives for the design of the smallish stations. Here we describe two alternatives to the smallish stations which we rejected as inferior to the design presented (by a small factor), but which may become attractive for VLSI technology parameters different from the ones we assumed. The availablity of alternatives with comparable costs gives further confidence in the soundness of our cost estimates.

C.1 Mesh-based emission routing

[`To be added.`]

C.2 Rotating stations

[`To be added.`]

D Relation to Previous Works

D.1 TWINKLE

As is evident from the presentation above, the new device shares with TWINKLE the property of time-space reversal compared to traditional sieving. Note, though, that this reversal is partially undone by the use of many parallel wires and some duplication of progression states. TWIRL is obviously faster than TWINKLE, because the two have comparable clock rates but the latter checks one sieve location per clock cycle whereas the former checks thousands. Since resources are shared by many bus lines, the increase in cost is much smaller than the run time reduction.

Also, the compact DRAM storage of largish progressions greatly reduces the device size (and it so happens that DRAM cannot be efficiently implemented on GaAs wafers, which are used by TWINKLE).

Whereas TWINKLE uses an analog optical adder, TWIRL reverts to digital electronic adder pipelines. We may consider replacing TWIRL's bus by optical adders, but constructing a separate optical adder for each residue class modulo s would entail practical difficulties, and would probably not be worthwhile because there are far fewer values to sum.

D.2 FPGA-Based Serial Sieving

Kim and Mangione-Smith [7] describe a sieving device using off-the-shelf parts that may be only 6 times slower than TWINKLE. It uses classical sieving, without time-memory reversal. The speedup follows from increased memory bandwidth – there are several FPGA chips and each is connected to multiple SRAM chips. As presented this implementation does not rival the speed or cost of TWIRL. Moreover, since it is tied to a specific hardware platform, it is unclear how it scales to larger parallelism and larger sieving problems.

D.3 Low-Memory Sieving Circuits

Bernstein [1] proposes to completely replace sieving by memory-efficient smoothness testing methods, such as the Elliptic Curve Method of factorization. These methods reduce the throughput cost of the matrix step from $y^{3+o(1)}$ to $y^{2+o(1)}$, where y is subexponential in the length of the integer being factored and depends on the choice of NFS parameters. By comparison, TWIRL has a throughput cost of $y^{2.5+o(1)}$, because the speedup factor grows as the square root of the number of progressions (cf. 4.5). However, these asymptotic figures hide significant factors; based on current experience, it appears unlikely that memory-efficient smoothness testing would rival the practical performance of traditional sieving for 1024-bit integers, let alone that of TWIRL, in spite of its superior asymptotic complexity.

D.4 Mesh-Based Sieving

While [1] deals primarily with the NFS matrix step, it does mention that the idea of "sieving via Schimmler's algorithm" and notes that its cost would be $L^{2.5+o(1)}$ (like TWIRL's). In [5], Geiselmann and Steinwandt follow this approach and give a detailed design for a mesh-based sieving circuit. The idea is to process a block of S sieve locations simultaneously by placing (p_i,r_i) pairs in a mesh, bringing together pairs with identical r_i values by snakelike sorting according to r_i, and then summing their $\lfloor \log p_i \rceil$ values locally.

When the number of progressions is x, the device of [5] handles $\Theta(x)$ sieve locations[21] in $\Theta(x^{1/2})$ clock cycles. By comparison, TWIRL handles s sieving locations in a single clock cycle, and $s = \Theta(x^{1/2})$. Thus, both achieve a speedup of $\Theta(x^{1/2})$. However, there are significant differences.

[21] Possibly less; an asymptotic analysis is lacking, especially in regard to the handling of small primes.

Duplication. In the design of [5], the state of each progression is duplicated $\lceil S/p_i \rceil$ times with $S = \Theta(x)$, or handled by other means. By comparison, in TWIRL we need to duplicate the smallish progressions only $s/\sqrt{p_i}$ times, where $s \approx \sqrt{S}$. This is quite significant. For the primary set of design parameters suggested in [5] 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: a separate division pipeline that tests potential candidates identified by the mesh, using over 12,000 expensive integer division units.

Generally, to keep the division pipeline at a reasonable size, the sums of $\lfloor \log p_i \rceil$ values over the progressions P_i handled by the mesh should be sufficiently well correlated with smoothness under all progressions. It is unclear how well this scales when all the small primes are omitted, especially given that the choice of NFS parameters ensures that there is a large contribution from small primes [11].

In TWIRL, the same progressions ($p_i < 2^{17}$) are handled efficiently by the tiny and small stations, thereby avoiding this problem. The rest of the progressions aren't duplicated at all.

DRAM storage. TWIRL's handling of largish primes using DRAM storage greatly reduces the size of the circuit when implemented using current VLSI technology (in [5] the same progressions would occupy about 2500 transistors each), making it possible to produce multiple devices per wafer even when factoring 1024-bit integers . It may be possible to adapt this to [5] (albeit only for $p_i \gg S \approx s^2$ rather than $p_i \gg s$), but this would yield little advantage because the duplication of primes smaller than S dominates cost. Also, putting large DRAM banks in mesh cells would incur some slowdown in routing operations due to propagation delays; in TWIRL this is not a problem, due to the unidirectional pipelined design.

Routing circuitry. TWIRL's delivery lines perform trivial one-dimensional unidirectional routing of values of size $\langle\!\langle 10+6 \rangle\!\rangle$ bits. This is much cheaper than [5]'s complicated two-dimensional mesh sorting of progression states of size $\langle\!\langle 2 \cdot 28 \rangle\!\rangle$.[22]

Inter-chip communication. If the device must span multiple wafers, the inter-wafer bandwidth requirements of our design are much lower than that of [5] (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.

Latency vs. throughput Finally, we note the following essential point. Mesh-based sorting (or routing) is effective in terms of *latency*, which is why it was appropriate for the matrix-step device of [1] where the input to each sorting operation 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.

Bottom line Due to the above, TWIRL is more efficient than [5] by a factor of 1600 for 512-bit composites (cf. 4.3), and this advantage can be expected to significantly increase for larger composites.

[22] The authors of [5] have suggested (in private communication) a variant of their device that routes emissions instead of sorting states, analogously to [10]. Still, mesh routing is more expensive than pipelined delivery lines.