Number crunching on clusters at EPFL

Joppe.Bos@epfl.ch, Marcelo Kaihara@epfl.ch, Thorsten.Kleinjung@epfl.ch, Arjen.Lenstra@epfl.ch & DagArne.Osvik@epfl.ch,
EPFL– School of Computer and Communication Sciences – Laboratory for cryptologic algorithms

Vous êtes vous déjà posé la question comment l'information est-elle protégée? Personne n'est vraiment sûr de cette protection. Aucune des méthodes habituelles de protection n'est garantie, tout ce qu'on peut dire c'est que nous ne pouvons pas les casser et on espère que d'autres ne le peuvent pas non plus. La sécurité à long terme repose sur des expériences comme celles décrites ici, menées dans le cadre du Laboratoire de cryptologie algorithmique de l'EPFL.

Ever wondered how information is protected? No one knows for sure. None of the currently used methods can be guaranteed to offer security. All we can say is that we cannot break them. We hope that others cannot do so either. Long term security estimates rely on experiments. Some of those carried out at EPFL's Laboratory for Cryptologic Algorithms are described.

Introduction

That 15 equals 3 times 5 is not hard to figure out.

That 2^{1039}-1 equals 5080711 times
55853 66661 99362 91260 74920 46583 15944 96864 65270 18488 63764 80100 52346 31985 32883 74753
times a 227-digit number is less obvious. Everyone with enough time (and patience) on their hands can verify it. But how were those numbers found? And why is it interesting?

Finding the factorizations of 15 or 2^{1039}-1 are examples of the *integer factorization problem*. It has been studied for ages, mostly for fun[1].

It was believed to be hard, and useless. The latter changed in 1976 when Ron Rivest, Adi Shamir, and Len Adleman showed an application. If it is hard, then everyone can communicate securely with anyone else. This now famous *RSA cryptosystem* led not only to headaches for national security agencies. It also put integer factorization in the center of attention. After more than three decades of scrutiny the results have been disappointing – and reassuring: integer factorization is still believed to be hard and RSA is still considered secure. And there is still no proof that the problem is hard either[2].

This is not the place to explain how the hardness of factoring can be used to protect information. We describe our experiments to find out *how large* an *RSA modulus* has to be to get enough protection. One of our experiments led to the factorization of 2^{1039}-1.

Only a few alternatives to RSA have been found. A popular one is ECC *(Elliptic Curve Cryptography)*. It relies on the hardness of ECDLP *(elliptic curve discrete logarithm problem)*. As in integer factorization, there is no hardness proof. But the problem looks even harder: secure ECC-parameters are much smaller than secure RSA-parameters.

Integer factorization and ECDLP experiments can be fully parallelized. Both require hundreds or even thousands of core years. For the rest they are entirely different. Integer factorization is a multi-step process. It profits from large memories and needs tightly coupled processors in one of its steps. Large clusters of servers are commonly used. For ECDLP almost anything goes, as long as there is a lot of computing power. It hardly needs memory and no fast network. It suffices to have a large disk to store the data trickling in from the contributors.

Our experiments were conducted on clusters at EPFL. For integer factoring this included the server clusters at LACAL *(Laboratory for Cryptologic Algorithms)* along with various other clusters (Callisto, Mizar, Pleiades) and the campus greedy network. For ECDLP we used LACAL's cluster of more than **200 PlayStation 3 game consoles**. We also describe some other cryptographic experiments on the PlayStation cluster.

[1] See for instance *Hunting big game in the theory of numbers*, a September 1932 **Scripta Mathematica** paper by Derrick N. Lehmer, cf. ***ed-thelen.org/comp-hist/Lehmer-NS03.html***.

[2] On the contrary, it is easy on a quantum computer. Such computers do not exist yet, so this is not a practical threat.

Generating RSA moduli

An RSA modulus is a publicly known integer that is the product of two prime numbers of about the same size. Security provided by it relies on the secrecy of its prime factors. RSA moduli can be generated quickly because of two classical results in number theory:

There are plenty of primes. About 1 out of every 2.3*D* random *D*-digit integers is prime. This is the *Prime Number Theorem*. If the random numbers are odd, the chance doubles!

Primes can quickly be recognized. If p is prime then $a^p - a$ is a multiple of p. This is *Fermat's little theorem*. A generalization is used to recognize primes.

Multiplication is easy. Twice using the above, two *D*-digit primes can efficiently be found. Their product can be calculated and made public. It is an RSA modulus of about 2D digits. The two primes should be kept secret by the *owner* of the RSA modulus.

If factoring is hard and *D* big enough, only the owner knows the factors of a public RSA modulus. With a good random number generator, different runs lead to different primes and different RSA moduli.

Number crunching on server clusters

Integer factorization

What can be hard about factoring? Just try to divide by 2, 3, 4, 5, 6, …. Or, faster, try only primes and stop at the target's square root. That works, except that it is slow. For 20-digit integers one may have to try almost half a billion primes. That is doable. For 100-digit numbers there may be more than 10^{45} primes to try, which is undoable. Nevertheless, 100-digit integers are easy to factor. How does that work?

Approximate factoring run times

To find the smallest factor p of a composite n, trial and error, Pollard's rho, ECM, SNFS, and NFS require, approximately,

$$\frac{p}{\ln p}, \quad \sqrt{p}, \quad e^{2\sqrt{\ln p \ \ln\ln p}}, \quad e^{1.56(\ln n)^{1/3}\ (\ln\ln n)^{2/3}}, \quad \text{and} \quad e^{1.92(\ln n)^{1/3}\ (\ln\ln n)^{2/3}}$$

operations on integers at most n, respectively. SNFS applies only to special n.

There are two types of factoring methods. The first type finds smaller factors faster. Examples are the above *trial and error* method, *Pollard's rho method*, and the *Elliptic Curve Method* (ECM, cf. Section *Slicing and dicing on the SPU*). For RSA moduli it is better to use NFS *(Number Field Sieve)*, the fastest method of the other type. SNFS, a faster version of NFS, can be used for *special* numbers. The number 2^{1039}-1 is special. RSA moduli are not.

ECM *(ecm.gforge.inria.fr/)* was installed in the spring of 2007 on EPFL clusters by two bachelor students, Aniruddha Bhargava and Sylvain Pelissier. Aniruddha is still managing it. More than 280 special numbers have been factored, as part of a century old factoring project ***(homes.cerias.purdue.edu/~ssw/cun/)***.

NFS on a desktop factors any 100-digit integer in a few hours. So, 100-digit integers are no good for RSA. Factoring

Factoring using relations

Let 143 be the number to be factored. Consider 17^2=3+2×143. We write it as

$$17^2 \equiv 3 \bmod 143$$

to express that the difference between 17^2 and 3 is an integer multiple of 143. We say that 17^2 and 3 are *congruent modulo* 143.

An integer $v > \sqrt{143}$ is a relation if $v^2 \equiv u \bmod 143$ and u's prime factors are at most 5: we say that u is *5-smooth*. Thus, 17 is a relation. Similarly, 19 is a relation because

$$19^2 \equiv 3{\times}5^2 \bmod 143.$$

But 18 is not, because $18^2 \equiv 38 \bmod 143$ and 38 has a prime factor 19 > 5.

Relations can be combined by multiplying them: the left hand side by the left hand side, and the right hand side by the right hand side. The result is again a relation. For instance, combining 17 and 19 produces

$$17^2{\times}19^2 \equiv 3{\times}3{\times}5^2 \bmod 143.$$

We seek a combination where both sides are squares. The left hand side is a product of squares and thus a square. For the right hand side it takes some fiddling around (linear algebra, done in the matrix step). For the example it turns out to be a square right away:

$$(17{\times}19)^2 \equiv (3{\times}5)^2 \bmod 143.$$

A square on both sides may be useful to factor:143 evenly divides

$$(17{\times}19)^2-(3{\times}5)^2=323^2-15^2=(323-15){\times}(323+15)=308{\times}338,$$

and therefore

$$143 = \gcd(143,308) \times \gcd(143,338)=11 \times 13.$$

Here we use the Euclidean algorithm to easily calculate the greatest common divisors.

200-digit integers with NFS is challenging. The first published 200-digit effort ran from Christmas 2003 to May 2005. It would have taken 75 years on a single core 2.2GHz Opteron. On that processor 300-digit numbers would take a million years. At this point that is out of reach for us. Currently, 309-digit RSA moduli are commonly used.

The two main steps of NFS are *sieving* and the *matrix*. Sieving is used to find *relations*: congruences that can be combined to produce a factorization. Sieving can be parallelized over any number of independent processors. The relations are combined in the matrix step. It needs all relations and is best run on tightly coupled processors. Below these steps are described in more detail.

Sieving

Relations in factoring use *smooth* values, integers with only small prime factors. Smooth integers are found quickly with a *sieve*: instead of checking each integer against all small primes, all values in the sieve are tested simultaneously. In theory this works great. In practice it is not so easy.

The sieve is too large to fit in memory: for 2^{1039}-1 the SNFS-sieve would have consisted of 10^{18} elements. Therefore the sieve is broken into pieces. *Line sieving* used to be the favorite approach. *Lattice sieving* with *special q* is faster and, right now, more popular. In both cases independent processors can process the smaller pieces. Because there are many small pieces, sieving can be parallelized over any number of processors. Cache misses cause trouble too. They cannot be avoided, but their impact can be lessened.

Sieving in NFS and SNFS

Let A and B be positive integers and let the sieve S be the rectangle $[-A,A]\times[1,B]$ in $\mathbf{Z}^2$ consisting of $\#S=(2A+1)B$ pairs of integers. Relations are pairs s in S for which $f(s)$ and $g(s)$ are smooth, where f and g are certain nicely behaving integer functions on $\mathbf{Z}^2$: if p divides $f(s_{1,0})$ for a pair $s_{1,0}=(r_f(p),1)$, then p also divides $f(s_{i,j})$ for all pairs $s_{i,j}=(ir_f(p)+jp,i)$ where i and j are integers. The same holds for g, but there is no relation between r_f and r_g. All $s_{i,j}$ in S are found by inspecting for each i with $1 \le i \le B$ the integers j for which $|ir_f(p)+jp| \le A$. After sieving twice with all small primes (namely, for $f(s)$ and for $g(s)$), the relations can be collected. The small primes are, approximately, those less than $\sqrt{\#S}$.

Line sieving

The sieve S can be split into smaller pieces $[-A,A]\times[i]$ for i=1,2,…,B without changing the sieving strategy. If 2A+1 sieve-locations still do not fit in memory, each line may be further partitioned.

Lattice sieving with special *q*

Given a prime q, let L_q be the lattice defined as the integer linear combinations of the vectors (q,0) and $(r_f(q),1)$ in $\mathbf{Z}^2$, and let S_q be a subset of L_q. It follows that q divides $f(s)$ for s in S_q. The lattice L_p may intersect with S_q. The intersection points are quickly determined using a reduced basis for the intersection of L_q and L_p. In S_q relations are found by doing this for all small p.

This lattice sieving touches only the pairs in S_q that are hit by p. Line sieving would for each p inspect each line of S_q. That would be too slow given how many primes q need to be processed and because the larger primes p hit a vanishingly small fraction of the lines of S_q.

The same relation may be found for different primes q. All duplicates need to be removed.

We give three sieving examples: the 200-digit record NFS factorization of RSA-200, the record SNFS factorization of 2^{1039}-1, and the current 232-digit NFS factoring effort for RSA-768.

RSA-200. Lattice sieving with most special q primes between 300 million and 1.1 billion was used, along with some line sieving, for small primes up to 300 million. It was done at various locations in Germany and the Netherlands, resulting in 2.3 billion unique relations. It would have taken 55 years on a single core 2.2 GHz Opteron with 1 GB RAM.

2^{1039}-1. The 40 million primes between 123 million and 911 million were used as special q primes. Per special q smoothness of twice 2 billion integers was tested using 16 million primes less than 300 million. This takes two and a half minutes on a single core of a 2.2GHz Opteron with 1GB RAM. For all special q primes it would have taken a century on a dual core 2.2GHz Opteron. It took half a year on clusters in Germany, Japan, and Switzerland. It was the first large scale factoring effort in which EPFL participated, contributing 8.3% of the sieving effort. More than 16 billion relations were collected including duplicates, resulting in almost 14 billion unique relations.

RSA-768. For this as-yet unfinished NFS factorization, half a billion special q primes in the range from 110 million to 11 billion sufficed. Per special q smoothness of twice 2 billion integers was tested using 55 million primes less than 1.1 billion. On average processing a single special q took a bit more than 2 minutes on a 2.2GHz Opteron core. Overall it would have taken about a millennium on a dual core 2.2GHz Opteron with 2GB RAM per core. It was done over a period of 1.5 years on clusters in Australia (0.43%), France (37.97%), Germany (8.14%), Japan (15.01%), Switzerland (34.33%), the Netherlands (3.44%), and the United Kingdom (0.69%).
Clusters at LACAL contributed 28.97% of the sieving effort, i.e., a sustained performance of about 200 dual core processors over a period of 1.5 years. Machines on EPFL's greedy campus network did not have enough RAM to contribute a lot. Nevertheless, they were responsible for 0.82%. More than 64 billion relations were found, resulting in 47 billion unique relations.

Relations and vectors

Looking for 7-smooth values while trying to factor 1457, we could have found the following five relations

$$41^2 \equiv 224 \bmod 1457=2^5\times3^0\times5^0\times7^1,$$
$$43^2 \equiv 392 \bmod 1457=2^3\times3^0\times5^0\times7^2,$$
$$58^2 \equiv 450 \bmod 1457=2^1\times3^2\times5^2\times7^0,$$
$$59^2 \equiv 567 \bmod 1457=2^0\times3^4\times5^0\times7^1,$$
$$60^2 \equiv 686 \bmod 1457=2^1\times3^0\times5^0\times7^3.$$

Since there are 4 primes that are at most 7, each relation leads to a 4-dimensional vector of exponents:

$$\begin{aligned}41&: [5,0,0,1],\\43&: [3,0,0,2],\\58&: [1,2,2,0],\\59&: [0,4,0,1],\\60&: [1,0,0,3].\end{aligned}$$

Component-wise adding the first, second, and fourth vector results in an all even vector:

$$[5,0,0,1]+[3,0,0,2]+[0,4,0,1]=[8,4,0,4].$$

This corresponds to the combination

$$(41\times43\times59)^2 \equiv (2^4\times3^2\times5^0\times7^2)^2 \bmod 1457.$$

With $41\times43\times59 \equiv 570 \bmod 1457$ and $2^4\times3^2\times7^2 \equiv 1228 \bmod 1457$ this leads to

$$1457= \gcd(1457,570-1228) \times \gcd(1457,570+1228)=47\times31.$$

Combination of the relations 41 and 60 produces the same factorization, but combination of 43 and 58 leads to 1457=1×1457. There is always a chance of bad luck. Sometimes many combinations have to be tried.

Sieving for 309-digit RSA moduli as used in practice is about a thousand times harder.

The matrix step

Each smooth value in a relation is the product of a number of small primes. For each smooth value, the number of times each small prime occurs in it, is the small prime's *exponent* - zero if the small prime does not occur. For each relation we get a *vector* of exponents. The number of exponents is the total number of small primes, and is the *dimension* of the vector.

In each smooth value each small prime can occur, but only very few small primes *do* occur. Thus, for each vector all but a few entries are zero, i.e., the vectors are *sparse*. Using simple sparse vector encoding tricks, the storage required for all vectors is therefore practically linear in the number of relations.

The set of relations thus leads to a collection of sparse vectors. In the matrix step subsets of the set of relations are determined such that the vectors corresponding to a subset add up, component-wise, to a vector with all even entries. This is a well known linear algebra problem. A solution exists if the number of relations exceeds the dimension. That condition is easy to check. In the examples above, however, there are way more unique relations than small primes. That is because *large primes* are allowed in smooth values. The dimension of the vectors – and the number of relations required – is therefore much larger than the number of small primes. It also makes it harder to see if enough relations have been found, i.e., if a solution exists. It is still easy, though. But existence of a solution is not enough. To be able to factor, solutions have to be found. That is a more complicated but well-studied problem.

The classical solution is *Gaussian elimination*. It processes the vectors one-by-one looking for a non-zero *pivot*, eliminating its occurrence in subsequent vectors. Although the original vectors require linear storage, it becomes quadratic due to *fill-in*. As a result the run time is cubic in the dimension, despite the original sparsity. For application in factoring, with dimensions of many millions, Gaussian elimination is too memory and time consuming.

Newer methods take advantage of the sparsity, with storage linear and run time quadratic in the dimension: *block-Lanczos* and *block-Wiedemann*. They look alike, as they both consist of a long iteration of matrix×vector multiplications. But they are very different. Lanczos is a geometric method that iteratively builds a sequence of orthogonal subspaces. After each iteration a central node has to gather all current information, to decide how to proceed for the next one. This frequent need for synchronization and non-trivial data exchange between all participating nodes limits the way block-Lanczos can be parallelized.

Block-Wiedemann is an algebraic method. It builds a sequence satisfying a linear recurrence relation, using an iteration of matrix×vector multiplications. The minimal polynomial of the recurrence, determined with the Berlekamp-Massey algorithm, is used to derive solutions using another iteration of matrix×vector multiplications.

The central step, i.e., Berlekamp-Massey, is the fastest one, but it requires lots of memory. The two iterations are the most compute-intensive. But they can be done independently by a small (say, 4, 8, 12, or 16) number of parties, so the brunt of the calculation can be divided among a small number of independent clusters. Inter-cluster communication is required only before and after the central step. Despite the cumbersome central step which is done at a single location, block-Wiedemann is now more popular than block-Lanczos: it is used in all current record factorizations.

RSA-200. After preprocessing, the relations resulted in 64 million vectors with, on average, 172 non-zeros per vector. The matrix step was done at BSI, Germany, in 3 months on a single cluster of 80 single core 2.2 GHz Opterons connected via a Gigabit network.

2^{1039}-1. The set of relations was squeezed down to 67 million vectors with 143 non-zeros on average. They could have been dealt with as the RSA-200 matrix, but it was decided to use a more challenging approach: this became the first factorization for which the matrix step was processed in 4 disjoint streams on clusters here at EPFL and at NTT in Japan.

At EPFL, a cluster of 96 2.66 GHz Dual Core2Duo processors (with 4 cores per node, sharing a single network connection) was used for 2 streams. At NTT 2 streams were processed on a cluster of 110 dual core 3GHz Pentium D processors in a torus topology with Gigabit ethernet. Under ideal circumstances all 4 streams could have been processed in 59 days on the Pentium cluster, i.e., 35 Pentium D core years. On 32 nodes of the Dual Core2Duo cluster it would have taken 162 days, i.e., 56 Dual Core2Duo core years. This latter performance is relatively poor due to the shared network connection.

The Berlekamp-Massey step was done on a 72 core cluster at EPFL. It took 128 GB of memory and less than 7 hours wall-clock time. On 64 cores at the University of Bonn it took 8 hours. Intermediate data transfer between NTT and EPFL took half a day over the Internet. Altogether the matrix step took 69 days.

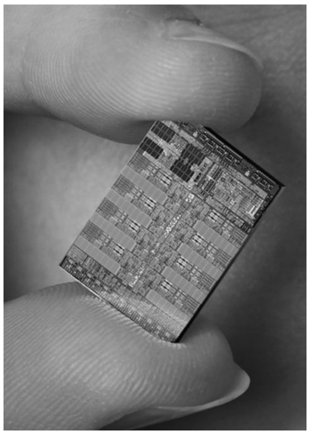
RSA-768. Here the matrix is much larger: 193 million vectors with 144 non-zeros on average. The calculation is ongoing on eight clusters here at EPFL (3), at INRIA/Nancy in France (3), and at NTT in Japan (2). The central step will be done on a cluster at EPFL. It would take four to five months on 36 nodes of LACAL's 12-cores-per-node 2.2 GHz AMD cluster with Infiniband network. On the NTT-cluster (as above), it would take about a year and a half. Combined (where INRIA uses **grid5000**, cf. ***www.grid5000.fr***), we hope to be able to do it in about 3 months.

For 309-digit RSA moduli as used in practice the matrix step is about a thousand times harder. A block-Lanczos effort may be carried out in the Netherlands on the Huygens computer, cf. ***huygens.supercomputer.nl***.

Number crunching on PlayStation 3 game consoles

The Cell processor

The Cell processor is the main processor of the Sony PlayStation 3 (PS3) game console. IBM's roadrunner, currently the largest computer, contains almost 13 thousand Cell processors – not to play games, but because they are powerful general purpose processors.On current PS3s the Cell can be accessed using Sony's hypervisor. The PS3 is thus a relatively inexpensive source of processing power.



The Cell (see picture) is quite different from regular server or desktop processors. Taking advantage of it requires new software. It is worthwhile to design software specifically for the Cell, because its architecture will soon be mainstream. It not only helps us to take advantage of inexpensive Cell processing power, it also helps to gear up for future processors.

The Cell's main processing power comes from eight SPUs *(Synergistic Processing Units)*. They run independently from each other at 3.2GHz, each working on their own 256 kilobyte of fast local memory (the *Local Store*) for instructions and data and their own 128 registers of 128 bits each. The latter allow SIMD *(Single Instruction Multiple Data)* operations on sixteen 8-bit, eight 16-bit, or four 32-bit integers. There are many boolean operations, but integer multiplication is limited to several 4-way SIMD 16×16 → 32-bit multipliers including a multiply-and-add. There is no 32×32 → 64-bit or 64×64 → 128-bit multiplier. The SPU has an odd and even pipeline: per clock cycle it can dispatch one odd and one even instruction. Because the SPU lacks smart branch prediction, branching is best avoided (as usual in SIMD). The Cell also has a PPE *(Power Processing Element)*, a dual-threaded 64-bit processor with 128-bit AltiVec/VMX SIMD unit.

When running Linux, six SPUs can be used (one is disabled, and one is reserved by the hypervisor). For some applications a Cell can be as powerful as twelve 64-bit processors or twice that many 32-bit ones. Sometimes we get more, sometimes less - but most of the time, even if integer multiplications are important, we get a lot.

Slicing and dicing on the SPU

We mostly looked at applications that can be run on any number of SPUs in parallel, on each individual SPU independent of the PPE or other SPUs, without inter-SPU communication, and without large memory demands. We have not tried hard yet to synchronize two or more SPUs for a single task. Per PS3 this would be doable, and could be efficient, if memory demands are low (i.e., probably not for NFS-sieving).

Our performance measure is overall throughput. Latency per process is mostly irrelevant. Given our applications' parallelizability over any number of SPUs, it may thus pay off to run several processes in parallel per SPU. While doing so, we may exploit the SIMD architecture by sharing instructions among processes. And we may interleave multiple SIMD processes, filling both pipelines to increase throughput, while possibly increasing per-process latency. It depends on the application and memory and code-size demands how

Discrete logarithms

Looking at 2^x mod 11 for x=0,1,2,…,9 we find that for each y with $1 \leq y < 11$ there is a unique x with $2^x \equiv y$ mod 11:

$$2^0 \equiv 1 \bmod 11,\ 2^1 \equiv 2 \bmod 11,\ 2^2 \equiv 4 \bmod 11,\ 2^3 \equiv 8 \bmod 11,\ 2^4 \equiv 5 \bmod 11,$$
$$2^5 \equiv 10 \bmod 11,\ 2^6 \equiv 9 \bmod 11,\ 2^7 \equiv 7 \bmod 11,\ 2^8 \equiv 3 \bmod 11,\ 2^9 \equiv 6 \bmod 11$$

(and $2^{10} \equiv 1$ mod 11). The integer 2 is said to *generate* the *multiplicative group of integers modulo* 11: <2> =($\mathbf{Z}/11\mathbf{Z}$)*. If $y \equiv 2^x$ mod 11, then x is *the discrete logarithm* of y with respect to 2 in ($\mathbf{Z}/11\mathbf{Z}$)*.

The element 3 of ($\mathbf{Z}/11\mathbf{Z}$)* *generates* an *order 5 subgroup* of ($\mathbf{Z}/11\mathbf{Z}$)*:

$$3^0 \equiv 1 \bmod 11,\ 3^1 \equiv 3 \bmod 11,\ 3^2 \equiv 9 \bmod 11,\ 3^3 \equiv 5 \bmod 11,\ 3^4 \equiv 4 \bmod 11,\ 3^5 \equiv 1 \bmod 11.$$

Given a prime p, a generator g of ($\mathbf{Z}/p\mathbf{Z}$)*, and an exponent x, the value $y \equiv g^x$ mod p in ($\mathbf{Z}/p\mathbf{Z}$)* can quickly be calculated using *modular exponentiation*, even for large p and x. The converse calculation, to find x such that $y \equiv g^x$ mod p if p, g, and y are given is the *discrete logarithm problem*. It is believed to be hard for large p. It is also believed to be hard in sufficiently large prime order subgroups of ($\mathbf{Z}/p\mathbf{Z}$)*.

The asymmetry in difficulty between modular exponentiation and its converse, is similar to the asymmetry between integer multiplication and factoring. Just as the latter underlies the RSA cryptosystem, the former underlies *discrete logarithm based cryptosystems*. Due to NFS-like discrete logarithm methods, the prime p in ($\mathbf{Z}/p\mathbf{Z}$)* would have to be as large as an RSA modulus to get the same level of security. Other methods may apply as well, depending on the order of the (sub)group used.

The birthday paradox

How many different people must be picked at random to get a more than 50% chance that two have the same birthday? It follows from a simple calculation that the answer is 23. This is called the *birthday paradox*, not because it is a paradox, but because 23 is much lower than intuition seems to suggest.

If random objects are selected with replacement from N objects, one may expect $\sqrt{\pi N/2}$ rounds before an object is picked twice. This *relatively high* chance to find a duplicate has many applications: in the search for hash collisions, in Pollard's rho for integer factoring, and Pollard's rho for discrete logarithms.

Pollard's rho method to compute discrete logarithms

Let g generate an order q group G, and let y be an element of G. To find the discrete logarithm of y with respect to g, i.e., an integer x such that $g^x = y$, Pollard's rho looks at $g^r y^s$ in G for random pairs (r,s) of integers. Because of the birthday paradox, after $\sqrt{\pi q/2}$ pairs a duplicate may be expected: pairs (r,s) and (u,v) such that $g^r y^s = g^u y^v$. Unless $v \equiv s$ mod q, this leads to $x = \dfrac{r-u}{v-s}$ mod q.

This is implemented by simulating a random walk in G. Take a small integer t (say, 15 or 20) and partition G in t parts G_1, G_2, …, G_t of about equal size, such that it can quickly be decided to which part an arbitrary element of G belongs. For i=1,2, …, t pick integers (r_i, s_i) at random and calculate $p_i = g^{r_i} y^{s_i}$ in G. Define the start point w_0 of the random walk as a random power of g. Given w_j, the walk's next point w_{j+1} is $p_i w_j$, where i is the unique integer such that w_j belongs to G_i. Note that it is easy to keep track of the pair (r,s) such that $w_j = g^r y^s$.

This is not a random walk, but close enough. To find duplicate points, use Floyd's cycle finding trick: compute (w_k, w_{2k}) for k=1,2,3, … until $w_k = w_{2k}$. On average this happens at $k \approx \sqrt{\pi q/2}$.

Parallelized Pollard rho with distinguished points

To parallelize Pollard's rho, different processes must be able to efficiently recognize if their walks hit the same point. To achieve this, each process generates a single random walk, each from a different random starting point, but all using the same partition of G and the same p_i for i=1,2, …,t. As soon as the walk hits upon a *distinguished point*, this point (along with its r and s) is reported to a central location, and the process starts a new walk from a new random starting point. A point is distinguished if a normalized representation of it has an easy to recognize property. This could be that l specific bits are zero, in which case walks may have average length 2^l (the choice depends on q, G, available disk space, etc.).

The idea is that if two walks collide – without noticing it – they will both ultimately reach the same distinguished point. This will be noticed after the points have been reported.

Elliptic curve groups

Elliptic curve groups are other groups where exponentiation (commonly referred to as *scalar multiplication* in elliptic curve context) is easy, and computing discrete logarithms is believed to be hard. No NFS-like tricks seem to apply. The parameters may therefore be chosen much smaller to reach adequate security. In practice they use *prime fields* or *binary extension fields*. The details are rather complicated.

The fastest method published to solve ECDLP is Pollard's rho with distinguished points.

many processes can profitably be squeezed together. The examples given below were run on LACAL's cluster of more than 215 PS3s (see picture), i.e., about 1300 SPUs.

Below i interleaved *j-way* SIMD processes on a single SPU is denoted by $i \times j$. If that is done sequentially $N>1$ times on the same SPU, we write $N \times (i \times j)$, for a total number of $N \times i \times j$ different and more or less simultaneous processes on a single SPU.

1×1: **non-interleaved single**. For applications where per-process latency must be minimized, it may be best to run a single process per SPU. If multi-precision integer arithmetic is needed, one could use IBM's off-the-shelf MPM library. Experiments with it did not meet our expectations. Our own modular arithmetic with integers up to 2048 bits outperforms *unrolled MPM* by a factor of at least two (the regular version by a bigger factor). Below parallelized approaches to multi-precision integer arithmetic are sketched. Per process they may be slower, but their throughput is better.

1×4: **non-interleaved 4-way SIMD**. We want to factor several numbers of the form 2^b-1, where b is an integer around 1200. This can be done with SNFS (cf. Section *Integer factorization*) at a huge effort per number. SNFS can be avoided if the number has small prime factors, because they can be found relatively quickly using ECM (the *Elliptic Curve Method* for integer factorization).

If we want to be reasonably confident to find factors of up to 65 digits, we must run 50 thousand ECM trials per number. Given a unique initial value per trial, each trial performs the same sequence of operations. We used 4-way SIMD integer arithmetic to process four trials simultaneously per SPU. This means that each variable occurs four times, with four values on which 4-way SIMD operations are carried out. When the values have b bits, each 128-bit register contributes at most 128/4=32 bits to each of the b-bit values.

For our range of b's, 4-way SIMD $b \times b \rightarrow 2b$-bit multiplication is split, using Karatsuba, into 4-way SIMD $320 \times 320 \rightarrow 640$-bit schoolbook multiplies. As this just fits in the SPU's 128 registers, there is no space to interleave multiple 4-way SIMD streams. Further optimizations or different approaches may change this. The modular reduction takes advantage of the special form of 2^b-1.

The first phase (with bound 3 billion) of four SIMD ECM trials takes about two days per SPU. With 1300 SPUs we need about three weeks for each 2^b-1 to process the first phase of 50 thousand ECM trials. The second phases will be done on regular clusters. This experiment started in September 2009. No factors have been found yet. For those 2^b-1 which fail to factor using ECM, we plan a new SNFS experiment. The 2^b-1's are not RSA moduli, but the resulting insights will be relevant for NFS and RSA moduli as well.

2×4: **doubly interleaved 4-way SIMD**. Cryptographic hashes are very different from RSA and ECC. They are used to *fingerprint* documents. Thus, it should be infeasible to find *collisions*: different documents with the same *hash*. In August 2004 collisions were published for the widely used hash function MD5. That this poses a practical threat was shown four years later with the proof-of-concept creation of a *rogue Certification Authority certificate*[3].

This was mostly done on the PS3 cluster. MD5 works on 32-bit values, so four MD5 hashes can be calculated simultaneously in 4-way SIMD mode. Two of such SIMD streams were interleaved. Taking advantage of the instruction set of the SPU, our 1300 SPUs performed as efficiently as eight thousand regular 32-bit cores.

50×(2×4): **multiple doubly interleaved 4-way SIMD**. We implemented Pollard's rho to compute a discrete logarithm in a 112-bit prime field elliptic curve group. The multi-precision integer arithmetic required for four random walks with distinguished points was implemented in the same 4-way SIMD fashion as used for the 2^b-1 ECM-application. But because the numbers here are much smaller, two 4-way SIMD walks were interleaved for added efficiency. Furthermore, this already 8-fold parallelism per SPU is further blown up by a factor 50 (by running it sequentially 50 times on the same SPU) for the following reason. To recognize distinguished points, each point on each walk must be normalized. Normalization in elliptic curve groups is not branch-free and not sympathetic to SIMD. Doing it for all 50×2×4 points would be too costly. At the cost of three additional 112-bit modular multiplications per walk, the normalizations were combined into a single one and the result divided again over the different walks. Per walk the high normalization cost is thus replaced by 1/400th of it plus three 112-bit modular multiplications. The 50 was the largest value for which all data would fit in the SPU's Local Store.

On 1300 SPUs this resulted in more than half a million parallel walks. Using increasingly efficient implementations it took half a year to find the desired result[4]. It is the current ECDLP record. With the latest version it would have taken three months and a half. More than half a billion distinguished points were generated with 24-bit distinguishing property. Their storage required 0.6 Terabytes of disk space. ECDLPs underlying prime

[3] See *www.win.tue.nl/hashclash/rogue-ca/*.

[4] See *lacal.epfl.ch/page81774.html*.

field ECC systems used in practice are about 20 million times harder to solve.

128×(2×1): **multiple doubly interleaved single**. We also implemented Pollard's rho to compute discrete logarithms in a 131-bit binary extension field elliptic curve group. Two 128-bit registers were used to represent a 131-bit value. Two walks were interleaved, and this was done sequentially 128 times per SPU to amortize the point-normalization cost. This resulted in 1/3 million parallel walks on 1300 SPUs. With a distinguished point probability of $2^{-35.4}$ and overall $2^{60.9}$ steps, $2^{25.5}$ distinguished points are needed. After one week 43'818 were found. This implies that the overall calculation can be expected to take 21 years on a cluster of 215 PS3s, or four and a half thousand years on a single PS3. If many others chip in, also using regular clusters, the calculation may be just about doable. ECDLPs underlying binary extension field ECC systems used in practice are about 20 thousand times harder to solve.

1×16: **non-interleaved 16-way SIMD**. A *block cipher* uses a *key* to encrypt a *block*, resulting again in a block. Block ciphers are used for high volume encryptions. AES *(Advanced Encryption Standard)* is the current standard block cipher. It uses keys of 128, 192, or 256 bits and blocks are 128-bit values.

We implemented AES *byte sliced* on the SPU, without further interleaving. In 3000 cycles an SPU simultaneously encrypts 16 blocks in SIMD fashion, all with the same 128-bit key. That is 11.7 cycles per byte. For decryption we get 14.4 cycles per byte. For batch encryption (decryption) with a single 128-bit key the SPU thus achieves 2.2 (1.8) Gigabit per second: in principle, a single PS3 can encrypt (decrypt) 1.65 (1.35) Gigabytes per second, using 6×16=96 parallel streams. For other key sizes the performance is similar.

1×128: **non-interleaved 128-way SIMD**. DES *(Data Encryption Standard)* is a block cipher with 56-bit keys and 64-bit blocks. Standardized in 1976, it is no longer considered secure. It was officially withdrawn in 2005, after the introduction of AES. It is still widely used and key search for DES is still relevant.

We designed a *bit sliced* DES-implementation for the SPU. It processes 128 keys in SIMD fashion, without further interleaving. Using 6 SPUs on a single PS3, a known plaintext DES key search should take, on average, 640 days. On the full cluster it becomes three days on average, and less than a week in the worst case.

Acknowledgements

This work was supported by the Swiss National Science Foundation under grant numbers 200021-119776 and 206021-117409 and by EPFL DIT. ■