HAL
open science

A Software Implementation of ECM for NFS

Alexander Kruppa

To cite this version:

Alexander Kruppa. A Software Implementation of ECM for NFS. [Research Report] RR-7041, INRIA. 2009. inria-00419094

HAL Id: inria-00419094

https://inria.hal.science/inria-00419094v1

Submitted on 22 Sep 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Software Implementation of ECM for NFS

Alexander Kruppa

N° 7041

Septembre 2009

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7041--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche NANCY - GRAND EST

A Software Implementation of ECM for NFS

Alexander Kruppa

Thème : Algorithmique, calcul certifié et cryptographie
Équipe-Projet CACAO

Rapport de recherche n° 7041 — Septembre 2009 — 39 pages

Abstract: The Elliptic Curve Method (ECM) of factorization can be used in the relation collection phase of the Number Field Sieve (NFS) to help identify smooth integers. This requires rapidly finding small prime factors for a large number of composites, each of a few machine words in size. We present a software implementation of ECM that is optimized for high throughput operation and compare it with recently proposed hardware implementations of ECM.

Key-words: Integer factoring, Elliptic Curves, Number Field Sieve

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Une implémentation d'ECM pour le crible algébrique

Résumé : L'algorithme ECM de factorisation d'entier est utilisé dans le crible algébrique (Number Field Sieve, NFS) pour identifier les relations friables. Cela nécessite de trouver rapidement des petits facteurs premiers dans un grand nombre d'entiers, chacun faisant quelques mots-machine. Nous présentons une implémentation logicielle d'ECM qui est optimisée pour ce cas précis et la comparons á des implémentations récentes en matériel.

Mots-clés : factorisation des entiers, courbes elliptiques, crible algébrique

1 Introduction

The sieving step of the Number Field Sieve [15] identifies integer pairs (a, b) with $a \perp b$ such that the values of two homogeneous polynomials $F_i(a, b)$, $i \in \{1, 2\}$, are both smooth, where the sieving parameters are chosen according to the smoothness criterion. Typically the two polynomials each have a "factor base bound" $\mathfrak{B}_i$, a "large prime bound" $\mathfrak{L}_i$, and a permissible maximum number of large primes k_i associated with them, so that $F_i(a, b)$ is considered smooth if it contains only prime factors up to $\mathfrak{B}_i$ except for up to k_i prime factors greater than $\mathfrak{B}_i$, but none exceeding $\mathfrak{L}_i$. For example, for the factorization of the RSA-155 challenge number [5] (a hard integer of 512-bit) the values $\mathfrak{B} = 2^{24}$, $\mathfrak{L} = 10^9$ and $k = 2$ were used for both polynomials. Kleinjung [14] gives an estimate for the cost of factoring a 1024-bit RSA key based on the parameters $\mathfrak{B}_1 = 1.1 \cdot 10^9$, $\mathfrak{B}_2 = 3 \cdot 10^8$, and $\mathfrak{L}_1 = \mathfrak{L}_2 = 2^{42}$ with $k_1 = 5$ and $k_2 = 4$.

The contribution of the factor base primes to each polynomial value $F_i(a, b)$ for a set of (a, b) pairs is approximated with a sieving procedure, which estimates roughly what the size of the polynomial values will be after factor base primes have been divided out. If these estimates for a particular (a, b) pair are small enough that both $F_i(a, b)$ values might be smooth, the polynomial values are computed, the factor base primes are divided out, and the two cofactors c_i are tested to see if they satisfy the smoothness criterion.

If only one large prime is permitted, no factoring needs to be carried out at all for the large primes: if $c_i > \mathfrak{L}_i$ for either i, this (a, b) pair is discarded. Since generally $\mathfrak{L}_i < \mathfrak{B}_i^2$ and all prime factors below $\mathfrak{B}_i$ have been removed, a cofactor $c_i \leq \mathfrak{L}_i$ is necessarily prime and need not be factored.

If up to two large primes are permitted, and the cofactor c_i is composite and therefore greater than the large prime bound but below $\mathfrak{L}_i^2$ (or a suitably chosen threshold somewhat less then $\mathfrak{L}_i^2$), it is factored. Since the prime factors in c_i are bounded below by $\mathfrak{B}_i$, and $\mathfrak{L}_i$ is typically less than $\mathfrak{B}_i^{1.5}$, the factors can be expected not to be very much smaller than the square root of the composite number. This way the advantage of special purpose factoring algorithms when small divisors (compared to the composite size) are present does not come into great effect, and general purpose factoring algorithms like SQUFOF or MPQS perform well. In previous implementations of QS and NFS, various algorithms for factoring composites of two prime factors have been used, including SQUFOF and Pollard-Rho in [9, chapter 3.6], and P–1, SQUFOF, and Pollard-Rho in [4, §3].

If more than two large primes are allowed, the advantage of special purpose factoring algorithms pays off. Given a composite cofactor $c_i > \mathfrak{L}_i^2$, we know that it can be smooth only if it has at least three prime factors, of which at least one must be less than $c_i^{1/3}$. If it has no such small factor, the cofactor is not smooth, and its factorization is not actually required, as this (a, b) pair will be discarded. Hence an early-abort strategy can be employed that uses special-purpose factoring algorithms until either a factor is found and the new cofactor can be tested for smoothness, or after a number of factoring attempts have failed, the cofactor may be assumed to be not smooth with high probability so that this (a, b) pair can be discarded.

Suitable candidates for factoring algorithms for this purpose are the P–1 method, the P+1 method, and the Elliptic Curve Method (ECM). All have in common that a prime factor p is found if the order of some group defined over

$\mathbb{F}_p$ is itself smooth. A beneficial property is that for ECM, and to a lesser extent for P+1, parameters can be chosen so that the group order has known small factors, making it more likely smooth. This is particularly effective if the prime factor to be found, and hence the group order, is small.

Although the P–1 and P+1 methods by themselves have a relatively poor asymptotic algebraic complexity in $O(\sqrt{p})$ (assuming an asymptotically fast stage 2 as described in [23] for example), they find surprisingly many primes in far less time, making them useful as a first quick try to eliminate easy cases before ECM begins. In fact, P–1 and P+1 may be viewed as being equivalent to less expensive ECM attempts (but also less effective, due to fewer known factors in the group order).

Another well-known special-purpose factoring algorithm is Pollard's "Rho" method [25] which looks for a collision modulo p in an iterated pseudo-random function modulo N, where p is a prime factor of N we hope to find. When choosing no less than $\sqrt{2\log(2)n} + 0.28$ integers uniformly at random from $[1, n]$, the probability of choosing at least one integer more than once is at least 0.5, well known as the Birthday Paradox which states that in a group of only 23 people, two share a birthday with more than 50% probablility. For the Rho method, the expected number of iterations to find a prime factor p is in $O\left(\sqrt{p}\right)$, and in case of Pollard's original algorithm, the average number of iterations for primes p around 2^{30} is close to $2^{15} \approx \sqrt{p}$, where each iteration takes three modular squarings and a modular multiplication, for an average of ≈ 130000 modular multiplications when counting squarings as multiplications. Brent [2] gives an improved iteration which reduces the number of multiplications by about 25% on average. We will see that a combination of P–1, P+1, and ECM does better on average.

Furthermore, trying the Pollard-Rho method with only a low number of iterations before moving on to other factoring algorithms has a negligible probability of success — among the 4798396 primes in $[2^{30}, 2^{30} + 10^8]$, only 3483 are found with at most 1000 iterations of the original Pollard-Rho algorithm with pseudo-random map $x \mapsto x^2 + 1$ and starting value $x_0 = 2$. For P–1, there are 1087179 primes p in the same range where the largest prime factor of $p - 1$ does not exceed 1000, and exponentiating by the product of all primes and prime powers up to B requires only $B/\log(2) + O\left(\sqrt{B}\right) \approx 1.44B$ squarings, compared to 4 multiplications per iteration for the original Pollard-Rho algorithm. By using a stage 2 for P–1, its advantage increases further. Figure 1 shows the distribution of the largest prime factor of $p - 1$ and the required number of Pollard-Rho iterations for finding p, respectively, for primes p in $[2^{30}, 2^{30} + 10^8]$. The distribution of the largest prime factor of $p + 1$ is identical to that of $p - 1$, up to statistical noise. We conclude that unlike P–1 and P+1, the Pollard-Rho method is not suitable for removing "easy pickings."

This research report describes an implementation of trial division for composites of a few machine words, as well as the P–1, P+1, and Elliptic Curve Method of factorization for small composites of one or two machine words, aimed at factoring cofactors as occur during the sieving phase of the Number Field Sieve.

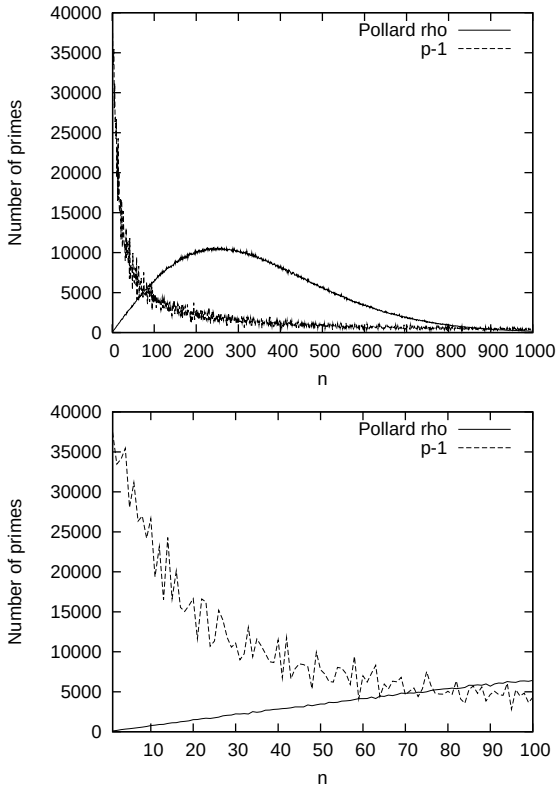

Figure 1: Number of primes p in $\left[2^{30}, 2^{30}+10^8\right]$ where the largest prime factor of $p-1$, respectively the number of Pollard-Rho iterations to find p, is in $[100n, 100n+99]$, $n \in \mathbb{N}$. The left graph shows $0 \leq n \leq 1000$, the right graph shows a zoom on $0 \leq n \leq 100$.

2 Trial Division

Before factoring of the non-sieved cofactor of the polynomial values into large primes can commence, the cofactor needs to be determined by dividing out all the factor base primes. For medium size factor base primes, say larger than a few hundred or a few thousand, a sieving technique ("re-sieving") can be used again that stores the primes when re-sieving hits a location previously marked as "likely smooth." For large factor base primes, say larger than a few ten thousand, the number of hits in the sieve area is small enough that the primes can be stored during the initial sieving process itself. For the smallest primes, however, re-sieving is inefficient, and a trial division technique should be used. This Section examines a fast trial division routine, based on ideas by Montgomery and Granlund [13] [21], that precomputes several values per candidate prime divisor to speed up the process.

2.1 Trial division algorithm

Given many composite integers N_i, $0 \le i < n$, we want to determine which primes from some set $P = \{p_j, 0 \le j < k\}$ of small odd primes divide each N_i. We assume $n \gg k$. Each N_i is a multi-word integer of up to $\ell + 1$ words, $N_i = \sum_{j=0}^{\ell} n_{i,j}\beta^j$, where β is the machine word base (e.g., $\beta = 2^{32}$ or $\beta = 2^{64}$) and ℓ is on the order of "a few," say $\ell \le 4$. For each prime $p \in P$, we precompute $w_j = \beta^j \bmod p$ for $1 \le j \le \ell$, $p_{\text{inv}} = p^{-1} \pmod{\beta}$ and $p_{\text{lim}} = \left\lfloor \frac{\beta-1}{p} \right\rfloor$.

Consider a particular integer $N = \sum_{j=0}^{\ell} n_j\beta^j$, and a particular prime $p \in P$. The algorithm first does a semi-reduction modulo p to obtain a single-word integer congruent to $N \pmod{p}$, then tests this single-word integer for divisibility by p.

To do so, we compute $r = n_0 + \sum_{j=1}^{\ell} n_j w_j \le (\beta - 1)(\ell(p-1) + 1)$. To simplify the next steps, we require $p < \sqrt{\frac{\beta}{\ell}}$. Even for $\beta = 2^{32}$, $\ell = 4$, this gives $p < 32768$ which is easily sufficient for trial division in NFS.

With this bound on p, we have $r < (\beta-1)(\sqrt{\beta\ell}-\ell+1)$. We then decompose r into $r = r_1\beta + r_0$, where $0 \le r_0 < \beta$. This implies $r_1 < \sqrt{\beta\ell}$, and $r_1 w_1 \le r_1(p-1) < \sqrt{\beta\ell}\left(\sqrt{\frac{\beta}{\ell}} - 1\right) = \beta - \sqrt{\beta\ell}$.

The algorithm then does another reduction step by $s = r_1 w_1 + r_0$. We would like $s = s_1\beta + s_0 < 2\beta - p$, so that a final reduction step $t = s_0 + s_1 w_1 < \beta$ produces a one-word result. Since $r_1(p-1) < \beta - \sqrt{\beta\ell}$, $s < 2\beta - \sqrt{\beta\ell} - 1 < 2\beta - p$. Since s_1 is either 0 or 1, the multiplication and addition in $s_0 + s_1 w_1$ is really just a conditional addition.

Now we have a one-word integer t which is divisible by p if and only if N is. To determine whether $p \mid t$, we use the idea from [13, §9] to compute $u = tp^{-1} \bmod \beta$, using the precomputed $p_{\text{inv}} = p^{-1} \pmod{\beta}$. If $p \mid t$, t/p is an integer $< \beta$ and so the modular arithmetic mod β must produce the correct $u = t/p$. There are $\left\lfloor \frac{\beta-1}{p} + 1 \right\rfloor$ multiples of p (including 0) less than β, under division by p these map to the integers $\left[0, \ldots, \left\lfloor \frac{\beta-1}{p} \right\rfloor\right]$. Since p is coprime to β, multiplication by $p^{-1} \pmod{\beta}$ is a bijective map, so all non-multiples of p must map to the remaining integers $\left[\left\lfloor \frac{\beta-1}{p} \right\rfloor + 1, \beta - 1\right]$. Hence the test for divisibility

can be done by a one-word multiplication by the precomputed constant p_{inv}, and one comparison to the precomputed constant $p_{\text{lim}} = \left\lfloor \frac{\beta-1}{p} \right\rfloor$.

2.2 Implementation

The algorithm is quite simple to implement on an x86 CPU, which offers the two-word product of two one-word arguments by a single MUL instruction. It might run as shown in Algorithm 1, where x_1, x_0 are registers that temporarily hold two-word products. A pair of registers holding a two-word value $r_1\beta + r_0$ is written as $r_1 : r_0$. The values $r_{0,1}$, $s_{0,1}$, and t_0 can all use the same registers, written $r_{0,1}$ here. The loop over j should be unrolled.

Input: Length ℓ
  $N = \sum_{i=0}^{\ell} n_i\beta^i$, $0 \le n_i < \beta$
  Odd prime $p < \sqrt{\frac{\beta}{\ell}}$
  $w_j = \beta^j \bmod p$ for $1 \le j \le \ell$
  $p_{\text{inv}} = p^{-1} \bmod \beta$
  $p_{\text{lim}} = \left\lfloor \frac{\beta-1}{p} \right\rfloor$
Output: 1 if $p \mid N$, 0 otherwise
$r_0 := n_0$;
$r_1 := 0$;
for $1 \le j \le \ell$ **do**
  $x_1 : x_0 = n_j \cdot w_j$;
  $r_1 : r_0 = r_1 : r_0 + x_1 : x_0$;
$x_0 = r_1 \cdot w_1$;
$r_0 = (r_0 + x_0) \bmod \beta$;
if *last addition set carry flag* **then**
  $r_0 = (r_0 + w_1) \bmod \beta$;
$r_0 = r_0 \cdot p_{\text{inv}}$;
if $r_0 \le p_{lim}$ **then**
  return 1;
else
  return 0;

Algorithm 1: Pseudo-code for trial division of numbers of up to $\ell+1$ words.

This code uses ℓ multiplications of two words to a two-word product. These multiplications are independent of one another, so they can overlap on a CPU with pipelined multiplier. On an Athlon64, Opteron, and Phenom CPUs, a multiplication can start every 2 clock cycles, the low word of the product is available after 4 clock cycles, the high word after 5 clock cycles. Thus in case of $\ell = 4$, the latency for the first 4 products and building their sum should be 12 cycles. The two remaining multiplies, the additions and conditional moves should be possible in about 11 cycles, giving a theoretical total count of about 23 clock cycles for trial dividing a 5 word integer by a small prime. Data movement from cache may introduce additional latency.

2.3 Use in NFS

Given a sieve region of size s with every d-th entry a sieve report, trial dividing by the prime p for all sieve reports has cost $O(s/d)$, while resieving has cost

$O(rs/p)$, where r is the number of roots modulo p the sieved polynomial has. Hence whether trial division or resieving is preferable will depend on $\frac{p}{dr}$, where those p with $\frac{p}{dr} < c$ for some threshold c should use trial division.

As primes are divided out of N, the number of words in N may decrease, making the following trial division faster. It might be worthwhile to try to reduce the size of N as quickly as possible. The probability that a prime p divides N may be estimated as r/p, the size decrease as $\log(p)$, so the probability that trial division by p will decrease the number of words in N may be estimated as being proportional to $r \log(p)/p$. For trial division, the candidate divisors p can be sorted so that this estimate is decreasing. This probability estimate does not take into account the fact that N, being a sieve report, is likely smooth, and under this condition the probability that p divides N increases by Bayes' theorem, more so for larger p than for small ones.

2.4 Testing several primes at once

Algorithm 1 reduces the input number to a one-word integer which is congruent to $N \pmod{p}$, then tests divisibility by p of that one-word integer. It is possible to do the reduction step for composite candidate divisors q, then test divisibility of the resulting one-word integer for all $p \mid q$. This way, for integers consisting of several words, the expensive reduction needs to be done only once for each q, the relatively cheap divisibility test for each p. This is attractive if the bound $q < \sqrt{\beta/\ell}$ is not too small. With $w = 2^{64}$, $\ell = 4$, we can use $q < 2147483648$, which allows for several small primes in q. For integers N with a larger number of words, it may be worthwhile to introduce an additional reduction step (for example, using Montgomery's REDC for a right-to-left reduction) to relax the bound on q to, e.g., $q < w/\ell$, so that the number of primes in q can be doubled at the cost of only two additional multiplies. In NFS, if the primes found by re-sieving have been divided out already before trial division begins, the N_i may not be large enough to make this approach worthwhile.

2.5 Performance of trial division

To measure the performance of the trial division code, we divide 10^7 consecutive integers of $1, \ldots, 5$ words by the first $n = 256, 512, 1024$, and 2048 odd primes on a 2 GHz AMD Phenom CPU, see Figure 2. The higher timings per trial division for $n = 256$ are due to the additional cost of dividing out found divisors, which has a greater relative contribution for smaller primes which divide more frequently. The timing for $\ell = 4, n = 2048$ is close to the predicted 23 clock cycles. The sudden increase for $n = 2048$ in the case of N with one word is due to caching: with 7 stored values (p, p_{inv}, p_{lim}, $w_{1,\ldots,4}$) of 8 bytes each, $n = 2048$ has a table of precomputed values of size 112kB, which exceeds the level-1 data cache size of 64kB of the Phenom. For large sets of candidate primes, the sequential passes through the precomputed data cause frequent misses in the level-1 cache, and the trial divisions for N of only one word are fast enough that transfer rate from the level-2 cache limits the execution. This could be avoided by computing fewer w_i constants (i.e., choosing a smaller ℓ) if the N are known to be small, or storing the w_i in separate arrays rather than interleaved, so that the w_i for larger i do not occupy cache while the N processed are small. Since the value of p is not actually needed during the trial division, it is possible to

	Number of words in N				
n	1	2	3	4	5
256	6.8 (2.6)	15.3 (6.0)	20.8 (8.1)	27.5 (10.7)	32.4 (12.6)
512	11.3 (2.2)	28.2 (5.5)	38.8 (7.6)	52.0 (10.2)	61.32 (12.0)
1024	21.3 (2.1)	54.9 (5.4)	75.9 (7.4)	102.0 (10.0)	120.7 (11.8)
2048	85.4 (4.1)	108.4 (5.3)	149.8 (7.3)	200.8 (9.8)	237.8 (11.6)

Figure 2: Time in seconds for trial division of 10^7 consecutive integers by the first n odd primes on a 2 GHz AMD Phenom CPU. Time per trial divison in nanoseconds in parentheses.

avoid storing it and recomputing it, e.g., from p_{inv} when it needs to be reported as a divisor.

3 Modular arithmetic

The modular arithmetic operations are relatively inexpensive when moduli and residues of only a few machine words are considered, and should be implemented in a way that lets the compiler perform in-lining of simple arithmetic functions to avoid unnecessary function call overhead and data movement between registers, memory and stack due to the calling conventions of the language and architecture. Many simple arithmetic operations can be implemented easily and efficiently using assembly language, but are cumbersome to write in pure C code, especially if multi-word products or carry propagation are involved. The GNU C compiler offers a very flexible method of injecting assembly code into C programs, with an interface that tells the compiler all constraints on input and output data of the assembly block so that it can perform optimization on the code surrounding the assembly statements. By defining some commonly used arithmetic operations in assembly, much of the modular arithmetic can be written in C, letting the compiler handle register allocation and data movement. The resulting code is usually not optimal, but quite useable. For the most time-critical operations, writing hand-optimized assembly code offers an additional speed improvement.

For the present work, modular arithmetic for moduli of 1 machine word and of 2 machine words with the two most significant bits zero is implemented. Implementation of arithmetic for moduli of 3 machine words is in progress.

3.1 Assembly support

To give an example of an elementary function that is implemented with the help of some assembly code, we examine modular addition with a modulus of 1 machine word. This is among the most simple operations possible, but useful as an example.

Let a "reduced residue" with respect to a positive modulus m mean an integer representative $0 \leq r < m$ of the residue class $r \pmod{m}$. Modular addition of

two reduced residues can be defined as

$$(a+b) \bmod m = \begin{cases} a+b-m & \text{if } a+b-m \geq 0 \\ a+b & \text{otherwise.} \end{cases}$$

If any modulus $m < \beta$ is permitted, where β is the machine word base, then the problem that $a+b$ might overflow the machine word arises. One could test for this case, then test if $a+b \geq m$, and subtract m if either is true, but this necessitates two tests. With a slight rearrangement, we can do with one:

1 $r := a + b$;
2 $s := a - m$;
3 $t := s + b$;
4 **if** *last addition set carry flag* **then**
5 r := t;

All arithmetic in this code is assumed modulo the word base β, i.e., the integers in r, s, and t are reduced residues modulo β. In line 2, since a is reduced modulo m, the subtraction $a - m$ necessarily produces a borrow, so that $s = a - m + \beta$. In line 3, if $s + b < \beta$, then this addition does not produce a carry, and $t = a + b - m + \beta < \beta$, i.e., $a + b - m < 0$. If $s + b \geq \beta$, the addition does produce a carry, and $0 \leq t = s + b - \beta = a + b - m$. Hence t is the proper result if and only if a carry occurs in line 3, to make up for the borrow of line 2. Lines 1 and 2 are independent and can be executed in parallel, leading to a dependent chain of length 3. We require $a < m$ for correctness, if $b \geq m$, the result still satisfies $r \equiv a + b \pmod{m}$ and $r < b$, but not necessarily $r < m$.

The implementation in C with a GCC x86 assembly block shown below. The value of s, shown separately for clarity above, is stored in `t` here.

```
r = a + b;
t = a - m;

__asm__ (
  "add %2, %1\n\t"    /* t := t + b */
  "cmovc %1, %0\n\t"  /* if (carry) r := t */
  : "+r" (r), "+&r" (t)
  : "g" (b)
  : "cc"
);
```

The computation of the initial `t` and `r` are done in C, to give the compiler some scheduling freedom. Since C does not provide direct access to the carry flag, the addition $t := t+b$ and the following conditional assignment are done in assembly. The constraints on the data passed to the assembly block state that the values of `r` and `t` must reside in registers (`"r"`) since the target of the conditional move instruction `cmovc` must be a register, and at least one of source or target of the addition instruction `add` must be a register. We allow the variable `b` to be passed in a register, in memory or as an immediate operand (`"g"`, "general" constraint, for x86_64 the correct constraint is `"rme"` since immediate constants are only 32 bit wide), which is the source operand to the `add` instruction. The `"+"` modifier tells that the values in `r` and `t` will be modified, and the `"&"` modifier tells that `t` may be modified before the end of the assembly block and thus no

other input variable should be passed in the register assigned to `t`, even if their values are known to be identical. Finally, `"cc"` tells the compiler that the values of the flags register may change. These constraints provide the information the compiler needs to be able to use the assembly block correctly, while leaving enough flexibility that it can optimize register allocation and data movement, compared to, e.g., compilers that require all parameters to assembly blocks in a fixed set of registers.

An alternative solution is to compute $r := b - (m - a)$ and adding m if the outer subtraction produced a borrow. However, this requires a conditional addition rather than a conditional move.

Similar to the modular addition, various functions such as modular subtraction and multiplication for one and two-word moduli, two-word addition, subtraction, multiplication and binary shift, and division with a two-word dividend (used, for example, for preparing a residue for use with REDC modular reduction with a two-word modulus, see 3.2) are written as functions with assembly support. As optimization effort progresses, more time-critical functions currently written in C with assembly macros will be replaced by dedicated assembly code.

3.2 Modular reduction with REDC

Montgomery presented in [17] a method for fast modular reduction. Given an integer $0 \leq a < \beta m$, for odd modulus m of one machine word and machine word base β (here assumed a power of 2), and a precomputed constant $m_{\text{inv}} = -m^{-1} \bmod \beta$, it computes an integer $0 \leq r < m$ which satisfies $r\beta \equiv a \pmod{m}$. It does so by computing the minimal non-negative tm such that $a + tm \equiv 0 \pmod{\beta}$, to make use of the fact that division by β is very inexpensive. Since $t < \beta$, $(a + tm)/\beta < 2m$, and at most one final subtraction of m ensures $r < m$. He calls the algorithm that carries out this reduction "REDC," shown in Algorithm 2.

Input: m, the modulus
β, the word base
$a < \beta m$, integer to reduce
$m_{\text{inv}} < \beta$ such that $m m_{\text{inv}} \equiv -1 \pmod{\beta}$
Output: $r < m$ with $r\beta \equiv a \pmod{m}$

$t := a \cdot m_{\text{inv}} \bmod \beta$;
$r := (a + t \cdot m)/\beta$;
if $r \geq m$ **then**
  $r := r - m$;

Algorithm 2: Algorithm REDC for modular reduction with one-word modulus. All variables take non-negative integer values.

The reduced residue output by this algorithm is not in the same residue class mod m as the input, but the residue class gets multiplied by $\beta^{-1} \pmod{m}$ in the process. To prevent accumulating powers of $\beta^{-1} \pmod{m}$ and having unequal powers of β when, e.g., adding or comparing residues, any residue modulo m is converted to Montgomery representation first, by multiplying it by β and reducing (without REDC) modulo m, i.e., the Montgomery representation of a

residue $a \pmod m$ is $a\beta \pmod m$. This way, if two residues in Montgomery representation $a\beta \pmod m$ and $b\beta \pmod m$ are multiplied and reduced via REDC, then $\mathrm{REDC}(a\beta b\beta) \equiv ab\beta \pmod m$ is the product in Montgomery representation. This ensures the exponent of β in the residues always stays 1, and so allows addition, subtraction, and equality tests of residues in Montgomery representation. Since $\beta \perp m$, we also have $a\beta \equiv 0 \pmod m$ if and only if $a \equiv 0 \pmod m$, and $\gcd(a\beta, m) = \gcd(a, m)$. Since $\beta = 2^{32}$ or 2^{64} is an integer square, the Jacobi symbol satisfies $\left(\frac{a\beta}{m}\right) = \left(\frac{a}{m}\right)$.

For moduli m of more than one machine word, say $m < \beta^k$, a product of two reduced residues may exceed β, but is below $m\beta^k$. The reduction can be carried out in two ways: one essentially performs the one-word REDC reduction k times, performing $O(k^2)$ one-word multiplies, the other replaces arithmetic modulo β in REDC by arithmetic modulo β^k, performing $O(1)$ k-word multiplications. In either case, a full reduction with (repeated one-word or a single multi-word) REDC divides the residue class of the output by β^k, and the conversion to Montgomery representation must multiply by β^k accordingly. The former method has lower overhead and is preferable for small moduli, the latter can use asymptotically fast multiplication algorithms if the modulus is large. As in our application the moduli are quite small, no more than two machine words, we use the former method.

Before modular arithmetic with REDC for a particular m can begin, the constant m_{inv} needs to be computed. If β is a power of 2, Hensel lifting makes this computation very fast. To speed it up further, we try to guess an approximation to m_{inv} so that a few least significant bits are correct, thus saving a few Newton iterations. The square of any odd integer is congruent to $1 \pmod 8$, so $m_{\mathrm{inv}} \equiv m \pmod 8$. The fourth bit of m_{inv} is equal to the binary exclusive-or of the second, third, and fourth bit of m, but on many microprocessors an alternative suggestion from Montgomery [22] is slightly faster: $(3m)$ XOR 2 gives the low 5 bits of m_{inv} correctly. Each Newton iteration $x \mapsto 2x - x^2 m$ doubles the number of correct bits, so that with either approximation, 3 iterations for $\beta = 2^{32}$ or 4 for $\beta = 2^{64}$ suffice.

Converting residues out of Montgomery representation can be performed quickly with REDC, but converting them to Montgomery representation requires another modular reduction algorithm. If such conversions are to be done frequently, it pays to precompute $\ell = \beta^2 \bmod m$, so that $\mathrm{REDC}(a\ell) = a\beta \bmod m$ allows using REDC for the purpose.

In some cases, the final conditional subtraction of m in REDC can be omitted. If $a < m$, then $a + tm < m\beta$ since $t < \beta$, so $r = (a + tm)/\beta < m$ which can be used when converting residues out of Montgomery form, or when division by a power of 2 modulo m is desired.

3.3 Modular inverse

To compute a modular inverse $r \equiv a^{-1} \pmod m$ for a given reduced residue a and odd modulus m with $a \perp m$, we use a binary extended Euclidean algorithm. Modular inverses are used at the beginning of stage 2 for the P–1 algorithm, and for initialisation of stage 1 of ECM (except for a select few curves which have simple enough parameters that they can be initialised using only division by small constants). Our code for a modular inverse takes about 0.5μs for one-word moduli, which in case of P–1 with small B_1 and B_2 parameters accounts

for several percent of the total run-time, showing that some optimization effort is warranted for this function.

The extended Euclidean algorithm solves

$$ar + ms = \gcd(a, m)$$

for given a, m by initialising $e_0 = 0, f_0 = 1, g_0 = m$ and $e_1 = 1, f_1 = 0, g_1 = a$, and computing sequences e_i, f_i and g_i that maintain

$$ae_i + mf_i = g_i \tag{1}$$

where $\gcd(a, m) \mid g_i$ and the g_i are strictly decreasing until $g_i = 0$. The original Euclidean algorithm uses $g_i = g_{i-2} \bmod g_{i-1}$, that is, in each step we write $g_i = g_{i-2} - g_{i-1}\lfloor \frac{g_{i-2}}{g_{i-1}} \rfloor$ and likewise $e_i = e_{i-2} - e_{i-1}\lfloor \frac{g_{i-2}}{g_{i-1}} \rfloor$ and $f_i = f_{i-2} - f_{i-1}\lfloor \frac{g_{i-2}}{g_{i-1}} \rfloor$, so that equation (1) holds for each i. If n is the smallest i such that $g_i = 0$, then $g_{n-1} = \gcd(a, m)$, $s = f_{n-1}$, and $r = e_{n-1}$. Since we only want the value of $r = e_{n-1}$, we don't need to compute the f_i values. We can write $u = e_{i-1}, v = e_i, x = g_{i-1}, y = g_i$ and for $i = 1$ initialise $u = 0, v = 1, x = m$, and $y = a$. Then each iteration $i \mapsto i + 1$ is computed by

$$(u, v, x, y) := (v, u - \lfloor x/y \rfloor v, y, x - \lfloor x/y \rfloor y).$$

At the first iteration where $y = 0$, we have $r = u$ and $x = 1$ if a and m were indeed coprime.

A problem with this algorithm is the costly computation of $\lfloor x/y \rfloor$ as integer division is usually slow. The binary extended Euclidean algorithm avoids this problem by using only subtraction and division by powers of 2. Our implementation is inspired by code written by Robert Harley for the ECCp-97 challenge and is shown in Algorithm 3. The updates maintain $ua \equiv -x2^t \pmod{m}$ and $va \equiv y2^t \pmod{m}$ so that when $y = 1$, we have $r = v2^{-t} = a^{-1} \pmod{m}$.

Input: Odd modulus m
Reduced residue $a \pmod{m}$
Output: Reduced residue $r \pmod{m}$ with $ar \equiv 1 \pmod{m}$, or failure if $\gcd(a, m) > 1$

if $a = 0$ **then**
  return *failure*;
$t := \mathrm{Val}_2(a)$; /* $2^t \mid\mid a$ */
$u := 0; v := 1; x := m; y := a/2^t$;
while $x \neq y$ **do**
  $\ell := \mathrm{Val}_2(x - y)$; /* $2^\ell \mid\mid x - y$ */
  if $x < y$ **then**
    $(u, v, x, y, t) := (u2^\ell, u + v, x, (y - x)/2^\ell, t + \ell)$;
  else
    $(u, v, x, y, t) := (u + v, v2^\ell, (x - y)/2^\ell, y, t + \ell)$;
if $y \neq 1$ **then**
  return *failure*;
$r := v2^{-t} \bmod m$;

Algorithm 3: Binary extended GCD algorithm.

In each step we subtract the smaller of x, y from the larger, so they are decreasing and non-negative. Neither can become zero as that implies $x = y$

in the previous iteration, which terminates the loop. Since both are odd at the beginning of each iteration, their difference is even, so one value decreases by at least a factor of 2, and the number of iterations is at most $\log_2(am)$. In each iteration, $uy + vx = m$, and since x and y are positive, $u, v \leq m$ so that no overflow occurs with fixed-precision arithmetic.

To perform the modular division $r = v/2^{t_i}$, we can use REDC. While $t \geq \log_2(\beta)$, we replace $v := \mathrm{REDC}(v)$ and $t := t - \log_2(\beta)$. Then, if $t > 0$, we perform a variable-width REDC to divide by 2^t rather than by β by computing $r = (v + ((vm_{\mathrm{inv}}) \bmod 2^t)\, m)\, /2^t$ with $mm_{\mathrm{inv}} \equiv -1 \pmod{\beta}$. Since $v < m$, we don't need a final subtraction in these REDC.

If the residue a whose inverse we want is given in Montgomery representation $a\beta^k \bmod m$ with k-word modulus m, we can use REDC $2k$ times to compute $a\beta^{-k} \bmod m$, then compute the modular inverse to obtain the inverse of a in Montgomery representation: $a^{-1}\beta^k \equiv \left(a\beta^{-k}\right)^{-1} \pmod{m}$. This can be simplified by using the fact that the binary extended GCD computes $v = a^{-1}2^t$. If we know beforehand that $t \geq \log_2 \beta$, we can skip divisions by β via REDC both before and after the binary extended GCD. Let the function $t(x, y)$ give the value of t at the end of Algorithm 3 for coprime inputs x, y. It satisfies

$$t(x,y) = \begin{cases} 0 & \text{if } x = y \text{ (implies } x = y = 1), \\ t(x/2, y) + 1 & \text{if } x \neq y,\ 2 \mid x, \\ t(x - y, y) & \text{if } x > y,\ 2 \nmid x, \\ t(y, x) & \text{if } x < y,\ 2 \nmid x. \end{cases}$$

Assuming y odd, case 3 is always followed by case 2, and we can substitute case 3 by $t(x, y) = t((x - y)/2, y) + 1$. We compare the decrease of the sum $x + y$ and the increase of t. In case 2, $(x + y) \mapsto x/2 + y > (x + y)/2$, and t increases by 1. In the substituted case 3, $(x + y) \mapsto (x + y)/2$, and t increases by 1. We see that whenever $x + y$ decreases, t increases, and whenever t increases by 1, $x + y$ drops by at most half, until $x + y = 2$. Hence $t(x, y) \geq \log_2(x + y) - 1$, and therefore $t(x, y) \geq \log_2(y)$, since $x > 0$.

Thus in case of k-word moduli $\beta^{k-1} < m < \beta^k$, we have $t(x, m) \geq (k - 1)\log_2(\beta)$ for any positive x, so using $a\beta^{-1} \pmod{m}$ as input to the binary extended GCD is sufficient to ensure that at the end we get $a^{-1}\beta \equiv v2^{-t} \pmod{m}$, or $a^{-1}\beta^k \equiv v2^{-t+(k-1)\log_2(\beta)} \pmod{m}$ and the desired result $a^{-1}\beta^k$ can be obtained from $v2^{-t}$ with a division by $2^{t-(k-1)\log_2(\beta)}$ via REDC.

3.4 Modular division by small integers

Initialisation of P+1 and ECM involves division of residues by small integers such as $3, 5, 7, 11, 13$ or 37. These can be carried out quickly by use of dedicated functions. To compute $r \equiv ad^{-1} \pmod{m}$ for a reduced residue a with $d \perp m$, we first compute $t = a + km$, with k such that $t \equiv 0 \pmod{d}$, i.e., $k = a\left(-m^{-1}\right) \bmod d$, where $-m^{-1} \bmod d$ is determined by look-up in a precomputed table for the $d - 1$ possible values of $m \bmod d$.

For one-word moduli, the resulting integer t can be divided by d via multiplication by the precomputed constant $d_{\mathrm{inv}} \equiv d^{-1} \pmod{\beta}$. Since $t/d < m < \beta$ is an integer, the result $r = td_{\mathrm{inv}} \bmod \beta$ produces the correct reduced residue r. This implies that computing t modulo β is sufficient.

For two-word moduli, we can choose an algorithm depending on whether m and d are large enough that t may overflow two machine words or not. In either case, we may write $t = t_1\beta + t_0$ with $0 \le t_0 < \beta$, $0 \le t_1 < d\beta$ and $r = r_1\beta + r_0$ with $0 \le r_0, r_1 < \beta$, and can compute $r_0 = t_0 d_{\text{inv}} \bmod \beta$.

If t does not overflow, we may write $t = t'' + t'd\beta, 0 \le t'' < d\beta$, where $d \mid t''$. Then $r = t/d = t'\beta + t''/d$ with $t''/d < \beta$, so we can compute $r_1 = \lfloor t_1/d \rfloor$. The truncating division by the invariant d can be implemented by the methods of [13]. An advantage of this approach is that the computation of the low word r_0 from t_0 is independent of the computation of the high word r_1 from t_1.

If t may overflow two machine words, we can compute r_0 as before, and use that $t - dr_0$ is divisible by $d\beta$, so we may write $r_1\beta + r_0 \equiv t/d \pmod{\beta^2}$ as $r_1 \equiv (t - dr_0)/\beta \cdot d_{\text{inv}} \pmod{\beta}$.

4 P–1 algorithm

The P–1 algorithm is described, for example, in [23]. We recapitulate some elementary facts here. The first stage of P–1 computes

$$x_1 = x_0^e \bmod N$$

for some starting value $x_0 \not\equiv 0, \pm 1 \pmod{N}$ and a highly composite integer exponent e. By Fermat's little theorem, if $p - 1 \mid e$ for any $p \mid N$, then $x_1 \equiv 1 \pmod{p}$ and $p \mid \gcd(x_1 - 1, N)$. This condition is sufficient but not necessary: it is enough (and necessary) that $\text{ord}_p(x_0) \mid e$, where $\text{ord}_p(x_0)$ is the order of x_0 in $\mathbb{F}_p^*$. To maximise the probability that $\text{ord}_p(x_0) \mid e$ for a given size of e, we could choose e to contain all primes and prime powers that divide $\text{ord}_p(x_0)$ with probability better than some bound $1/B_1$. One typically assumes that a prime power q^k divides $\text{ord}_p(x_0)$ with probability q^{-k}, so that e is taken as the product of all primes and prime powers not exceeding B_1, or $e = \text{lcm}(1, 2, 3, 4, \ldots, B_1)$.

The value of e is precomputed and passed to the P–1 stage 1 routine, which basically consists only of a modular exponentiation, a subtraction and a gcd. The base x_0 for the exponentiation is chosen as 2; in a left-to-right binary powering ladder, this requires only squarings and doublings, where the latter can be performed quickly with an addition instead of a multiplication by x_0.

To reduce the probability that all prime factors of N (i.e., N itself) are found simultaneously and reported as a divisor at the end of stage 1, only the odd part of e is processed at first, and then the factors of 2 in e one at a time by successive squarings. After each one we check if the new residue is 1 (mod N), indicating that all prime factors of N have been found now, and if so, revert to the previous value, to use it for the gcd. If not the same power of 2 divides $\text{ord}_p(x_0)$ exactly for all primes $p \mid N$, then this will discover a proper factor. This backtracking scheme is simple but satisfactorily effective: Among 10^6 composite numbers that occurred during an sieving experiment of the RSA155 number, each composite being of up to 86 bits and with prime factors larger than 2^{24}, only 48 had the input number reported as the factor in P–1 stage 1 with $B_1 = 500$. Without the backtracking scheme (i.e., processing the full exponentiation by e, then taking a GCD), 879 input numbers are reported as factors instead.

The second stage of P–1 can use exactly the same implementation as the second stage of P+1, by passing $X_1 = x_1 + x_1^{-1}$ to the stage 2 algorithm.

The stage 2 algorithm for ECM is very similar as well, and they are described together in Section 7.

4.1 P–1 stage 1 performance

Table 1 compares the performance of the P–1 stage 1 implementation for different B_1 values and modulus sizes on AMD Phenom and Intel Core 2 CPUs.

	Core 2		Phenom	
B_1	1 word	2 words −2 bits	1 word	2 words −2 bits
100	3.15	6.24	2.49	4.59
200	5.38	12.2	4.12	8.26
300	7.28	17.2	5.51	11.3
400	9.23	22.2	6.92	14.5
500	11.4	27.8	8.49	18.0
600	13.2	32.7	9.83	21.0
700	15.4	38.2	11.4	24.4
800	17.2	43.1	12.7	27.5
900	19.4	48.5	14.2	30.9
1000	21.4	53.8	15.7	34.1

Table 1: Time in microseconds for P–1 stage 1 with different B_1 values on 2.146 GHz Intel Core 2 and 2 GHz AMD Phenom CPUs.

5 P+1 algorithm

The P+1 algorithm is described in detail in [23]. We recapitulate the basic algorithm here.

The first stage of P+1 computes $x_1 = V_e(x_0) \bmod N$, where $x_0 \in \mathbb{Z}/N\mathbb{Z}$ is a parameter, $V_n(x)$ is a degree-n Chebyshev polynomial defined by $V_n(x+x^{-1}) = x^n + x^{-n}$, and e is a highly composite integer chosen as for the P–1 method. These Chebyshev polynomials satisfy $V_0(x) = 2$, $V_1(x) = x$, $V_{-n}(x) = V_n(x)$, $V_{mn}(x) = V_m(V_n(x))$, and $V_{m+n}(x) = V_m(x)V_n(x) - V_{m-n}(x)$.

We test for a factor by taking $\gcd(x_1 - 2, N)$. If there is a prime p such that $p \mid N$ and $p - \left(\frac{\Delta}{p}\right) \mid e$, where $\Delta = x_0^2 - 4$ and $\left(\frac{\Delta}{p}\right)$ is the Legendre symbol, then $p \mid \gcd(x_1 - 2, N)$.

Since V_{n-m} is required for computing V_{n+m}, these polynomials cannot be evaluated with a simple binary addition chain as in the case of the exponentiation in stage 1 of P–1. Instead, an addition chain needs to be used that contains $n - m$ whenever the sum $n + m$ is formed from n and m. These chains are described in Section 5.1.

The required addition chain for the stage 1 multiplier e is precomputed and stored as compressed byte code, see Section 5.2.

As for P–1, a backtracking scheme is used to avoid finding all factors of N and thus reporting the input number as the factor found. Since factors of 2 in e can easily be handled by $V_{2n}(x) = V_2(V_n(x)) = V_n(x)^2 - 2$, they need not be stored in the pre-computed addition chain, and can be processed one at a time. Similarly as in stage 1 of P–1, we remember the previous residue, process one

factor of 2 of e, and if the result is 2 (mod N), meaning that all factors of N have been found, we revert to the previous residue to take the GCD with N. Using the same 10^6 composite inputs as for P–1, P+1 with $B_1 = 500$ reports 117 input numbers as factors with backtracking, and 1527 without.

If stage 1 of P+1 is unsuccessful, we can try to find a factor yet by running stage 2, using as input the output x_1 of stage 1. Our stage 2 is identical for P–1 and P+1, and very similar for ECM, and is described in Section 7.

5.1 Lucas chains

Montgomery shows in [19] how to generate addition chains $a_0, a_1, \ldots, a_\ell$ with $a_0 = 1$ and length ℓ such that for any $0 < i \leq \ell$, there exist $0 \leq s, t < i$ such that $a_i = a_s + a_t$ and $a_s - a_t$ is either zero, or is also present in the chain. He calls such chains "Lucas chains." For example, the addition chain $1, 2, 4, 5$ is not a Lucas chain since the last term 5 can be generated only from $4 + 1$, but $4 - 1 = 3$ is not in the chain. The addition chain $1, 2, 3, 5$, however, is a Lucas chain. For any positive integer n, $L(n)$ denotes the length of an optimal (i.e., shortest possible) Lucas chain that ends in n.

A simple but generally non-optimal way of generating such chains uses the reduction $(n, n-1) \mapsto (\lceil n/2 \rceil, \lceil n/2 \rceil - 1)$. We can compute $V_n(x)$ and $V_{n-1}(x)$ from $V_{\lceil n/2 \rceil}(x)$, $V_{\lceil n/2 \rceil - 1}(x)$, $V_1(x) = x$, and $V_0(x) = 2$. In the case of n even, we use $V_n(x) = V_{\lceil n/2 \rceil}(x)^2 - V_0(x)$, and $V_{n-1}(x) = V_{\lceil n/2 \rceil}(x) V_{\lceil n/2 \rceil - 1}(x) - V_1(x)$ and in the case of n odd, we use $V_n = V_{\lceil n/2 \rceil}(x) V_{\lceil n/2 \rceil - 1}(x) - V_1(x)$. and $V_{n-1}(x) = V_{\lceil n/2 \rceil - 1}(x)^2 - V_0(x)$ The resulting chain allows processing the multiplier left-to-right one bit at a time, and thus is called binary chain by Montgomery. Each bit in the multiplier adds two terms to the addition chain, except that when processing the final bit, only one of the two values needs to be computed, and if the two most significant bits (MSB) are 10_b, the above rule would compute $V_2(x)$ twice of which one should be skipped. Any trailing zero bits can be handled by $V_{2n}(x) = V_n(x)^2 - V_0(x)$ at the cost of 1 multiplication each. The length $L_b(n2^k)$ for the binary Lucas chain for a number $n2^k$ with n odd is therefore $2\lfloor \log_2(n) \rfloor - 1 + k$ if the two MSB are 10_b, or $2\lfloor \log_2(n) \rfloor + k$ if $n = 1$ or the two MSB are 11_b. Examples are in Table 2. It lists the binary chain, the length $L_b(n)$ of the binary chain, an optimal chain, and the length $L(n)$ of an optimal chain, for odd n up to 15.

n	Binary chain	$L_b(n)$	Optimal chain	L(n)
$3 = 11_b$	$1, 2, 3$	2	$1, 2, 3$	2
$5 = 101_b$	$1, 2, 3, 5$	3	$1, 2, 3, 5$	3
$7 = 111_b$	$1, 2, 3, 4, 7$	4	$1, 2, 3, 4, 7$	4
$9 = 1001_b$	$1, 2, 3, 4, 5, 9$	5	$1, 2, 3, 6, 9$	4
$11 = 1011_b$	$1, 2, 3, 5, 6, 11$	5	$1, 2, 3, 5, 6, 11$	5
$13 = 1101_b$	$1, 2, 3, 4, 6, 7, 13$	6	$1, 2, 3, 5, 8, 13$	5
$15 = 1111_b$	$1, 2, 3, 4, 7, 8, 15$	6	$1, 2, 3, 6, 9, 15$	5

Table 2: Binary and optimal Lucas chains for small odd values n

The binary chain is very easy to implement, but produces non-optimal Lucas chains except for very small multipliers. The smallest positive integer where the binary method does not produce an optimal chain is 9, and the smallest such

prime is 13. Montgomery shows that if n is a prime but not a Fibonacci prime, an optimal Lucas chain for n has length $L(n) \geq r$ with r minimal such that $n \leq F_{r+2} - F_{r-3}$, where F_k is the k-th Fibonacci number. Since $F_k = (\phi^k - \phi^{-k})/\sqrt{5}$ where $\phi = (1+\sqrt{5})/2$ is the Golden Ratio, this suggests that if this bound is tight, for large n an optimal chain for n should be about 28% shorter than the binary chain.

In a Lucas chain $a_0, a_1, \ldots, a_\ell$ of length ℓ, a doubling step $a_{k+1} = 2a_k$ causes all a_i with $k \leq i \leq \ell$ to be multiples of a_k, and all these terms a_i are formed using sums and differences only of terms $a_j, k < j \leq \ell$, see [19]. Such a doubling step corresponds to a concatenation of Lucas chains. For composite $n = n_1 \cdot n_2$, a Lucas chain can be made by concatenating the chains of its factors. E.g., for $n = 15$, we could multiply every entry in the chain $1, 2, 3, 5$ by 3 and append it do the chain $1, 2, 3$ (omitting the repeated entry 3) to form the Lucas chain $1, 2, 3, 6, 9, 15$. Since any Lucas chain starts with $1, 2$, every concatenation introduces one doubling step, and every doubling step leads to a chain that is the concatenation of two Lucas chains. Chains that are not the concatenation of other chains (i.e., that contain no doubling step other than $1, 2$) are called simple chains. For prime n, only simple chains exist. In the case of binary Lucas chains, the concatenated chain is never longer than the chain for the composite value and usually shorter, so that forming a concatenated Lucas chain from chains of the prime factors of n (if known) is always advisable. The same is not true for optimal chains, as shown below.

Optimal chains can be found by exhaustive search for a chosen maximal length $l_{\max}$ and maximal end-value $n_{\max}$. For odd $n \geq 3$, a Lucas chain for n always starts with $1, 2, 3$ since a doubling step $2, 4$ would produce only even values in the remainder of the chain. In the exhaustive search, the Lucas chain $a_0, \ldots, a_k$ can be extended recursively if $k < \ell_{\max}$ and $a_k < n_{\max}$ by adding an element $a_{k+1} > a_k$ such that the resulting sequence is still a Lucas chain, i.e., satisfying that there are $0 \leq i, j \leq k$ such that either $a_{k+1} = 2a_i$, or $a_{k+1} = a_i + a_j$ and $a_i - a_j$ is present in the chain. For each chain so created, we check in a table of best known lengths whether the length $k+1$ is smaller than the previously known shortest length for reaching a_{k+1}, and if so, update it to $k+1$ and store the current chain as the best known for reaching a_{k+1}. By trying all possible chain expansions, we are certain to find an optimal chain for every $n \leq n_{\max}$. This recursive search is very time consuming due to a large number of combinations to try. To reach a worthwhile search depth, the possible chain extensions can be restricted. The last step of an optimal chain is always $a_l = a_{\ell-1} + a_{\ell-2}$ as otherwise one or both of $a_{\ell-1}, a_{\ell-2}$ are obsolete, so the table of best known lengths needs to be checked and updated only after such an addition step, and the final recursion level of the search needs to consider only this addition step. Any doubling step $a_{k+1} = 2a_k$ causes the chain to become the equivalent of a concatenated chain, so during the recursive chain expansion, doubling steps need not be considered. Instead the recursive search produces only the optimal lengths of simple chains. Then for all possible pairs $3 \leq m \leq n \leq \sqrt{n_{\max}}$, the length of the chain for mn is updated with the sum of the lengths of chains for m and n, if the latter is shorter. This is repeated until no more improvements occur. After the first pass, the optimal lengths of chains for all n where n has at most two prime factors are known. After the second pass, for all n that contain at most three primes, etc., until after at most $O(\log(n_{\max}))$ passes optimal lengths for all values are known. Using this

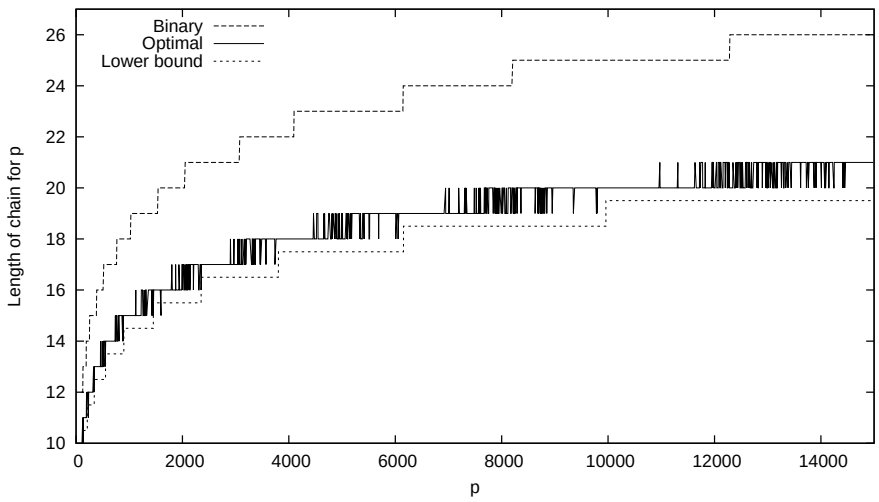

Figure 3: Length of binary and optimal Lucas chains for odd primes p in $[100, 15000]$, and a lower bound on the length for primes that are not Fibonacci primes. The graph for the bound is set 0.5 lower to make it visible. The Fibonacci prime 1597 is seen to undershoot this lower bound.

search method, the minimal lengths of Lucas chains for primes $100 < n < 10000$ have been determined, shown in Figure 3. It compares the length of the binary Lucas chain, the optimal Lucas chain and the lower bound on the length of Lucas chains for primes that aren't Fibonacci primes. This lower bound is quite tight, in the examined data $L(n)$ does not exceed it by more than 1. The Fibonacci prime 1597 visibly undershoots this lower bound (as does the smaller Fibonacci prime 233, but it is difficult to see in the graph).

The exhaustive search method is extremely slow and useless for producing addition chains for P+1 or ECM if large B_1 values are desired. Montgomery [19] suggests the algorithm "PRAC," which produces Lucas chains based on GCD chains, noting that a subtractive GCD algorithm for n, r with $n > r$ and $n \perp r$ always produces a valid Lucas chain for n. However, the resulting Lucas chain has length equal to the sum of the partial quotients in the continued fraction expansion of $n/(n-r)$, and if a large partial quotient appears, the resulting Lucas chain is unreasonably long. He fixes this problem by introducing additional rules for reduction in the GCD chain (rather than just replacing the larger of the two partial remainders by their absolute difference as in a purely subtractive GCD chain) to avoid situations where the quotient of the partial remainders deviates too far from the Golden Ratio, yet satisfying the conditions for a Lucas chain. The great advantage is that PRAC usually produces very good chains and does so rapidly. This way it is feasible to try a few different suitable r for a given n, and for n in the range of interest for P+1 and ECM, one usually discovers an optimal chain this way.

It remains the problem of choosing a suitable $r \perp n$ to start the GCD chain, hoping to find a (near) optimal chain. Montgomery suggests trying $r = n - \lfloor n/c \rceil$ for several irrational c such that the continued fraction expansion of c has small

partial quotients. This way, the partial fraction expansion of $n/(n-r)$ starts with small partial quotients as well. Good choices are the golden Ratio $c_0 = \phi$, whose partial quotients all are 1, or numbers with partial quotients all 1 except for one or two 2 among the first 10 partial quotients. The resulting large number of multipliers is not a problem if the Lucas chains are precomputed, but in cases where they are computed on-the-fly during stage 1 of P+1 or ECM, a smaller set of multipliers should be used, say, only those with at most one 2 among the first ten partial quotients.

Even with a large set of c_i values to try, PRAC in the form given by Montgomery cannot always obtain an optimal chain. The smallest example is $n = 751$ which has two Lucas chains of optimal length $L(751) = 14$:
$1, 2, 3, 5, 7, 12, 19, 24, 43, 67, 110, 177, 287, 464, 751$ and
$1, 2, 3, 5, 8, 13, 21, 34, 55, 68, 123, 191, 314, 437, 751$.
Both chains involve an addition step that references a difference that occurred 5 steps before the new term: for the former sequence in the step $a_8 = 43 = a_7 + a_6 = 24 + 19$, with difference $a_7 - a_6 = 5 = a_3$, and for the latter sequence in the step $a_{10} = 123 = a_9 + a_8 = 68 + 55$, with difference $a_9 - a_8 = 13 = a_5$. The original PRAC algorithm does not have any rule that allows utilizing a difference that occurred more than 4 steps before the new term and so cannot find either of these two chains. Another, similar case is $n = 1087$. For primes below 10000, I found 40 cases where PRAC did not find an optimal chain. For the purpose of generating Lucas chains for P+1 and ECM, these missed opportunities at optimal chains are of no great consequence. When using P+1 and ECM as a factoring subroutine in NFS, the B_1 value is often less than 751 so that such cases do not occur at all, and if a greater B_1 should be used, they occur so rarely that adding more rules to PRAC so that optimal chains are found for all primes below B_1 would increase the code complexity of our P+1 or ECM stage 1, which implements each PRAC rule (see Section 5.2), for little gain. For our implementation, this was not deemed worthwhile. For the purpose of finding optimal Lucas chains rapidly, it would be interesting to augment PRAC with a suitable rule for the required addition step $a_k = a_{k-1} + a_{k-2}$ with $a_{k-1} - a_{k-2} = a_{k-5}$, and testing which primes remain such that the modified PRAC cannot find optimal chains.

For composite $n = pq$, we trivially have $L(n) \leq L(p) + L(q)$, since we can concatenate the chain for p and the chain for q. In some cases, a shorter chain for the composite n exists than for the concatenated chains of its factors. The smallest example is $1219 = 23 \cdot 53$ which has

$$1, 2, 3, 4, 7, 11, 18, 29, 47, 76, 123, 170, 293, 463, 756, 1219$$

as an optimal chain of length 15, while an optimal chain for 23 is $1, 2, 3, 4, 5, 9, 14, 23$ of length 7, and for 53 is $1, 2, 3, 5, 6, 7, 13, 20, 33, 53$ of length 9.

Similarly, composite numbers n exist where PRAC with a certain set of multipliers finds a chain for n that is shorter than the concatenated simple chains for the divisors of n. A problem is that the starting pair n, r for the GCD sequence must be coprime, possibly making several c multipliers ineligible for an n with small prime factors. Starting with a large enough set of multipliers, usually enough of them produce coprime n and r that an optimal chain can be found, if one exists of a form suitable for PRAC. The example $n = 1219$ above is found, e.g., with $r = 882$, using the multiplier $3 - \Phi$ with continued fraction expansion $//1, 2, 1, 1, 1, 1, \ldots//$.

5.2 Byte code and compression

In implementations of P+1 or ECM such as in GMP-ECM [29] that typically operates on numbers of hundreds to ten-thousands of digits, or in the ECM implementation of Prime95 [28] that operates on number of up to several million digits, the cost of generating good Lucas chains on-the-fly during stage 1 is mostly negligible, except for P+1 on relatively small numbers of only a few hundred digits. However, in an implementation of ECM and especially P+1 designed for numbers of only a few machine words, the on-the-fly generation of Lucas chains would take an inacceptable part of the total run-time. Since in our application of using P+1 and ECM as a factoring sub-routine in NFS, identical stage 1 parameters are used many times over again, it is possible to pre-compute optimized Lucas chains and process the stored chain during P+1 or ECM stage 1.

This raises the question how the chain should be stored. Since the PRAC algorithm repeatedly applies one of nine rules to produce a Lucas chain for a given input, an obvious method is to store the sequence of PRAC rules to apply. The precomputation outputs a sequence of bytes where each byte stores the index of the PRAC rule to use, or one of two extra indices for the initial doubling resp. the final addition that is common to all (near-)optimal Lucas chains. This way, a byte code is generated that can be processed by an interpreter to carry out the stage 1 computations for P+1 or ECM. For each prime to include in stage 1, the corresponding byte code is simply appended to the byte code, which results in a (long) concatenated Lucas chain for the product of all stage 1 primes. If primes are to be included whose product is known to have a better simple Lucas chain than the concatenation of the chains for the individual primes, then their product should be passed to the byte-code generating function.

The byte code generated by PRAC is highly repetitive. For example, byte codes for the PRAC chains for the primes 101, 103, 107, and 109 are

$$\begin{array}{ll} 101: & 10,3,3,0,3,3,0,5,3,3,3,11 \\ 103: & 10,3,0,3,3,0,3,3,0,4,3,11 \\ 107: & 10,3,0,3,3,0,3,0,4,3,3,3,11 \\ 109: & 10,3,0,3,0,1,1,3,11 \end{array}$$

It is beneficial to reduce redundancy in the byte code to speed up stage 1. The byte code interpreter that executes stage 1 must fetch a code byte, then call the program code that carries out the arithmetic operations that implement the PRAC rule indicated by the code; thus there is a cost associated with each code byte. If the interpreter uses a computed jump to the code implementing each PRAC rule, there is also a branch misprediction each time a code byte is different from the previous one, as current microprocessors typically predict computed jumps as going to the same address as they did the previous time. Some PRAC rules frequently occur together, such as rule 3 follwed by rule 0, so that merging them may lead to simplifications in the arithmetic. In particular, rules 11 (end of a simple chain) and 10 (start of a new simple chain) always appear together, except at the very start and at the very end of the byte code.

These issues are addressed by byte code compression. A simple static dictionary coder greedily translates frequently observed patterns into new codes. The byte code interpreter implements merged rules accordingly. For example, the byte code sequence "3, 0" (for an addition follwed by a swap of variable

contents) occurs very frequently and may be translated to a new code, say 12, and the interpreter performs a merged addition and swap. The codes "11, 10" always occur as a pair and can be substituted except at the very start and the very end of the bytecode, but these two occurrences can be hard-coded into the interpreter, so they do not need to be considered individually at all.

Since we often can choose among several different Lucas chains of equal length for a given stage 1 prime by using different multipliers in PRAC, we can pick one that leads to the simplest compressed code by compressing each candidate chain, and choosing the one that has the smallest number of code bytes and code byte changes.

For comparison, without any compression or effort to reduce the number of code bytes or code changes when choosing PRAC multipliers, the byte code for a stage 1 with $B_1 = 500$ consists of 1487 code bytes and 1357 code changes, whereas even with the simple substitution rules described above and careful choice of PRAC multipliers to minimize the number of code bytes and code changes, only 554 code bytes with 435 code changes remain.

5.3 P+1 stage 1 performance

Table 3 compares the performance of the P+1 stage 1 implementation for different B_1 values and modulus sizes on AMD Phenom and Intel Core 2 CPUs.

	Core 2		Phenom	
B_1	1 word	2 words −2 bits	1 word	2 words −2 bits
100	4.04	8.44	3.45	6.30
200	7.50	17.3	6.32	12.3
300	10.3	24.6	8.69	17.2
400	13.4	32.5	11.2	22.3
500	16.6	40.7	14.0	27.9
600	19.5	48.0	16.4	32.8
700	22.8	56.6	19.1	38.5
800	25.7	64.0	21.5	43.5
900	28.9	72.4	24.2	48.9
1000	32.0	80.4	26.7	54.2

Table 3: Time in microseconds for P+1 stage 1 with different B_1 values on 2.146 GHz Intel Core 2 and 2 GHz AMD Phenom CPUs, using precomputed Lucas chains stored as compressed byte code.

For comparison, without using byte code compression or choosing the PRAC multipliers to minimize byte code length and number of code changes, on Core 2, P+1 stage 1 with 1 word and $B_1 = 500$ takes $20.4\mu s$ and so is about 22% slower, and with 2 words takes $50.4\mu s$ and so is about 24% slower.

6 ECM algorithm

The Elliptic Curve Method of factorization was introduced by H. W. Lenstra in 1987 [16]. Whereas P−1 works in the group $\mathbb{F}_p^*$ of order $p-1$ and P+1 in a subgroup of $\mathbb{F}_{p^2}^*$ of order $p-1$ or $p+1$, ECM works in the Mordell-Weil group of points on an elliptic curve defined over $\mathbb{F}_p$. By Hasse's theorem, the number

of points and therefore the order of the Mordell-Weil group of an elliptic curve over $\mathbb{F}_p$ is in $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$. The number of points on a particular curve depends on both the curve parameters and the field. ECM finds a prime factor p of N if the curve over F_p has smooth order; the advantage of ECM over previous algorithms such as P–1 and P+1 (which always work in a group of order $p-1$ or $p+1$) is that many different curves can be tried, until one with sufficiently smooth order comes along.

Any elliptic curve E over a field K of characteristic neither 2 nor 3 can be defined by the Weierstraß equation

$$y^2 = x^3 + ax + b. \tag{2}$$

This equation defines an elliptic curve if and only if the discriminant $4a^3 + 27b^2$ does not vanish. The set of points on E consists of the solutions $(x, y) \in K^2$ of (2), plus the point at infinity $\mathcal{O}$.

The group addition law of two points on the curve is defined geometrically by putting a straight line through the two points (or, if the points are identical, the tangent of the curve in that point), taking the line's intersection point with the curve and mirroring it at the x-axis. Since the curve is symmetric around the x-axis, the mirrored point is on the curve, and is the result. If the straight line is vertical, no intersection point exists; in this case the point at infinity is taken as the result. The point at infinity is identity element of the group, adding it to any point results in the same point. The inverse of a point is the point mirrored at the x-axis. This addition law on the points of an elliptic curve defines an Abelian group, see for example [26].

The Weierstraß form of elliptic curves can be used for implementing ECM, but requires a costly modular inverse in the computation of point additions. Montgomery [18] proposes an alternative form of elliptic curve equation in projective coordinates so that its addition law avoids modular inverses, while still keeping the number of required multiplications low. His curves are of form

$$BY^2Z = X(X^2 + AXZ + Z^2), \tag{3}$$

with points $(X : Y : Z) \in K^3$ satisfying (3), where X, Y, Z are not all zero. Two points are identical if $(X_2 : Y_2 : Z_2) = (kX_1 : kY_1 : kZ_1)$ for some $k \in K$, $k \neq 0$. The point at infinity is $\mathcal{O} = (0 : 1 : 0)$.

Not all elliptic curves over finite fields can be brought into form (3), but we may restrict our ECM implementation to use only these curves. Montgomery describes an addition law for curves of this form. Given two distinct points P_1 and P_2, we can compute the X and Z-coordinates of $P_1 + P_2$ from the X and Z-coordinates of P_1, P_2 and $P_1 - P_2$. Similarly, we can compute the X and Z-coordinates of $2P$ from the X and Z-coordinates of P and the curve parameters. Surprisingly, the Y-coordinate is not needed in these computations, and can be ignored entirely when using curves in Montgomery form for ECM, and points are commonly written as only $(X :: Z)$ with Y-coordinate omitted. The details of the addition law are found in [18, 10.3.1] or [20, 2.3].

This addition law requires that in order to form the sum of two points, their difference is known or zero. This is reminiscent of the P+1 method where we need $V_{m-n}(x)$ to compute $V_{m+n}(x)$ from $V_m(x)$ and $V_n(x)$, and the same Lucas chains used to compute $V_k(x)$ for integer k in P+1 can be used to compute the multiple kP of a point P on a curve in Montgomery form in ECM.

6.1 ECM stage 1

In stage 1 of ECM, we choose a suitable curve E of form (3) defined over $\mathbb{Z}/N\mathbb{Z}$, where N is the integer we wish to factor. Naturally N is composite, so $\mathbb{Z}/N\mathbb{Z}$ is a ring but not a field, but this has little consequence for the arithmetic of the curve as the only operation that could fail is inversion of a ring element, and an unsuccessful inversion of a non-zero element in $\mathbb{Z}/N\mathbb{Z}$ reveals a proper factor of N which is the exact purpose of ECM. We often consider the curve E_p for a prime $p \mid N$, which is the curve E reduced modulo p, i.e., E over the field $\mathbb{F}_p$.

We then choose a point P_0 on E and compute $P_1 = e \cdot P_0$ for a highly composite integer e, usually taken to be divisible by all primes and prime powers up to a suitably chosen value B_1, i.e., $e = \mathrm{lcm}(1, 2, 3, 4, \ldots, B_1)$. We hope that for some prime factor p of N, the order of P_0 on E_p is B_1-smooth (and thus divides e), since then the point P_1 on E_p will be the point at infinity $(0 :: 0)$ so that P_1 has Z-coordinate $0 \pmod p$ and $p \mid \gcd(P_Z, N)$.

To find a point P_0 on E over $\mathbb{Z}/N\mathbb{Z}$, we choose a point of E over $\mathbb{Q}$ and map it to $\mathbb{Z}/N\mathbb{Z}$. The point over $\mathbb{Q}$ must not be a torsion point, or P_0 will have identical order on E_p for all $p \mid N$ so that P_1 is the point at infinity either for all E_p or for none, producing only the trivial factorizations N or 1.

By careful selection of the curve E we can ensure that number of points of E_p is a multiple of 12 or 16, significantly increasing the probability that the order of P_0 is smooth. The choice of E is described in Section 6.2.

The computation of $P_1 = e \cdot P_0$ on E is carried out by use of pre-computed Lucas chains, similarly as in the P+1 algorithm. The selection of near-optimal Lucas chains for ECM is described in Section 6.3.

If stage 1 of ECM is unsuccessful, we try stage 2 where we hope to find a prime p such that the order of P_0 on E_p factors into primes and prime powers up to B_1, except for one bigger (but not too much bigger) prime q. Our stage 2 is very similar for P–1, P+1, and ECM and is described in Section 7.

6.2 Choice of curve

In a letter to Richard Brent, Hiromi Suyama [27] showed that curves of form (3) over $\mathbb{F}_p$ always have group order divisible by 4, and also showed a parametrization that ensures that the group order is divisible by 12, which Brent describes in [3]. This parametrization generates an infinite family of curves over $\mathbb{Q}$ which can be used to generate a large number of distinct curves modulo N. For a given integer parameter $\sigma \neq 0, 1, 3, 5$, let

$$\begin{aligned} &u = \sigma^2 - 5, v = 4\sigma, \\ &X_0 = u^3, Z_0 = v^3 \text{ and } A = \tfrac{(v-u)^3(3u+v)}{4u^3 v} - 2. \end{aligned} \tag{4}$$

Then the point $(X_0 :: Z_0)$ is on the curve (3) with parameter A. The same parametrization is used by GMP-ECM [29, 1] and Prime95 [28].

Montgomery showed in his thesis [20] how to choose curves of form 3 such that the curve over $\mathbb{Q}$ has a torsion subgroup of order 12 or 16, leading to group order divisible by 12 or 16, respectively, when the curve is mapped to $\mathbb{F}_p$ for almost all p.

For curves with rational torsion group of order 12 he uses

$$\begin{aligned} &t^2 = \tfrac{u^2-12}{4u}, a = \tfrac{t^2-1}{t^2+3} \\ &X_0 = 3a^2 + 1, Z_0 = 4a \text{ and } A = \tfrac{-3a^4-6a^2+1}{4a^3}, \end{aligned} \tag{5}$$

where $u^3 - 12u$ is a rational square. The solutions of $v^2 = u^3 - 12u$ form an elliptic curve of rank 1 and 2-torsion over $\mathbb{Q}$, with generator $(-2, 4)$ and 2-torsion point $(0, 0)$. However, adding the torsion point or not seems to produce isomorphic curves for ECM, so we ignore it. Hence for a given integer parameter $k > 1$ we can compute suitable values of u and v by computing $k \cdot (-2, 4)$ on $v^2 = u^3 - 12u$. We can then let $t = v/(2u)$. This produces an infinite family of curves over $\mathbb{Q}$.

Curves with torsion 16 and positive rank over $\mathbb{Q}$ are more difficult to generate, see [20, 6.2] for details. We currently implement only one such curve with $X_0 = 8$, $Z_0 = 15$, and $A = 54721/14400$.

These parametrizations ensure that the group order is divisible by 12 or 16, respectively, but the resulting group order of the curve over $\mathbb{F}_p$ does not behave like an integer chosen uniformly at random from the integers that are multiples of 12 or 16, respectively, in the Hasse interval around p. In particular, the average valuation of 2 in the group order for curves with rational torsion 12 is 11/3, slightly higher than 10/3 for curves in Brent-Suyama parametrization (which have rational torsion 6), making them somewhat more likely to find factors.

Very small σ-values for the Brent-Suyama parametrization lead to curves with simple rationals for the point coordinate and curve parameter, and very small k-values for Montgomery's parametrization for curves with rational torsion 12 lead to simple rationals for a, see Table 4. These rationals can be mapped to $\mathbb{Z}/N\mathbb{Z}$ easily, as the denominators are highly composite integers so that the required divisions modulo N can be done by the methods of Section 3.4 and a few multiplications.

When factoring cofactors after the sieving step of NFS into large primes, only very few curves are required on average since the primes to be found are relatively small, and with an early-abort strategy, only the first few curves work on larger composites where arithmetic is more expensive. In spite of the small number of curves with such simple rationals as curve parameters, it is useful to implement them as special cases.

σ	X_0	Z_0	A
2	-1	512	$-3645/32$
4	1331	4096	$6125/85184$

k	a	X_0	Z_0	A
2	$-3/13$	$196/169$	$-12/13$	$-4798/351$
3	$28/37$	$3721/1369$	$112/37$	$-6409583/3248896$

Table 4: Some elliptic curves chosen by the Brent-Suyama parametrization with group order divisible by 12, and by Montgomery's parametrization with rational torsion group of order 12.

6.3 Lucas chains for ECM

In principle, Lucas chains for ECM can be chosen exactly as for P+1. However, a subtle difference exists: in P+1, the cost of a doubling $V_{2n}(x) = V_n(x)^2 - 2$ is identical to that of an addition $V_{m+n}(x) = V_m(x)V_n(x) - V_{m-n}$ if V_{m-n} is

known and a squaring is taken to have the same cost as a multiplication. This way, the cost of a Lucas chain depends only on its length.

In ECM, the cost of a point doubling usually differs from the cost of an addition of distinct points. In the addition rules given by Montgomery, a doubling takes 5 modular multiplications of which 2 are squarings, whereas an addition of distinct points takes 6 modular multiplications of which again 2 are squarings.

These different costs can be taken into account when choosing Lucas chains. For example, to multiply a point by 7, we can choose between the chains $1, 2, 3, 5, 7$ or $1, 2, 3, 4, 7$ of equal length. In the former, all additions except for the initial doubling $1, 2$ are additions of distinct values. In the latter, we can produce 4 by doubling 2, so that this Lucas chain would save 1 modular multiplication in the elliptic curve arithmetic.

When generating Lucas chains with PRAC using several multipliers, we can choose the best chain not according to its length but by the cost of the arithmetic performed in each PRAC rule that is used to build the chain.

The speedup in practice is relatively small: with two-word modulus, ECM stage 1 with $B_1 = 500$ is about 1% faster when counting the cost of a doubling as 5/6 of the count of an addition when choosing Lucas chains. Still, this improvement is so simple to implement that it may be considered worthwhile.

As for P+1, the precomputed addition chains are stored as byte code that describes a sequence of PRAC rules to apply. Code compression may be used to reduce the overhead in the byte code interpreter, but since the elliptic curve arithmetic is more expensive than in the case of $P + 1$, the relative speedup gained by compression is much smaller.

6.4 ECM stage 1 performance

Table 5 compares the performance of the ECM stage 1 implementation for different B_1 values and modulus sizes on AMD Phenom and Intel Core 2 CPUs.

	Core 2		Phenom	
B_1	1 word	2 words −2 bits	1 word	2 words −2 bits
100	11.8	35.6	9.33	24.4
200	24.5	77.9	19.4	52.6
300	35.3	113	27.8	76.0
400	46.7	151	36.6	101
500	58.7	190	46.2	127
600	69.6	226	54.6	151
700	82.3	266	64.5	178
800	93.6	302	72.4	202
900	105	342	82.5	229
1000	117	381	92.0	255

Table 5: Time in microseconds for ECM stage 1 with different B_1 values on 2.146 GHz Core 2 and 2 GHz AMD Phenom CPUs

7 Stage 2 for P–1, P+1, and ECM

Stage 1 of P–1, P+1, and ECM all compute an element g_0^e of some (multiplicatively written) group G for a highly composite integer e, typically chosen as $e = \mathrm{lcm}(1, 2, 3, 4, \ldots, B_1)$ for some integer B_1. If the order of g_0 is B_1-smooth, then $g_1 = g_0^e$ is the identity in G. Since G is defined over F_p where p divides N, the number to factor, we can construct from the identity in G a residue $r \pmod N$ such that $r \equiv 0 \pmod p$ but hopefully not $r \equiv 0 \pmod N$, and then $\gcd(r, N)$ usually reveals p. If the order of g_0 is not B_1-smooth, stage 1 fails to find p. However, we may be able to find it yet if the order of g_0 consists of a B_1-smooth part times a not-too-large prime q.

Stage 2 of P–1, P+1, and ECM tries to find the value of q efficiently on the assumption that q is prime and not very large, although larger than B_1, by looking for a match $g_1^m = g_1^n$ which occurs when $q \mid m - n$. We will describe the stage 2 for the P+1 algorithm; P–1 can use the same algorithm by adjusting its output, and the stage 2 for ECM is structurally very similar. Differences between the P+1 and ECM stage 2 are noted.

Our stage 2 is modeled after the enhanced standard continuation described by Montgomery [18]. For given search limits B_1 and B_2 and input X_1 it chooses a value d with $6 \mid d$ and computes two lists

$$f_i = V_{id}(X_1) \bmod N \text{ for } \lfloor B_1/d \rceil \le i \le \lfloor B_2/d \rceil \text{ and} \tag{6}$$

$$g_j = V_j(X_1) \bmod N \text{ for } 1 \le j < d/2 \text{ and } j \perp d, \tag{7}$$

so that all primes q in $]B_1, B_2]$ can be written as $q = id - j$ or $q = id + j$.

Let $X_1 \equiv \alpha_1 + 1/\alpha_1 \pmod N$, where α_1 may be in a quadratic extension of $\mathbb{Z}/N\mathbb{Z}$, and assume

$$\alpha_1^q \equiv 1 \pmod p \tag{8}$$

for some unknown prime p, $p \mid N$ and a prime q, $B_1 < q \le B_2$. Let $q = id - j$ or $q = id + j$. Then, using $V_{-n}(X) = V_n(X)$, we have

$$\begin{aligned} V_{id}(X_1) \equiv V_{q\pm j}(X_1) &\equiv \alpha_1^{q\pm j} + 1/\alpha_1^{q\pm j} \\ &\equiv \alpha_1^{\pm j} + 1/\alpha_1^{\pm j} \equiv V_{\pm j}(X_1) \equiv V_j(X_1) \pmod p \end{aligned}$$

and so

$$V_{id}(X_1) - V_j(X_1) \equiv 0 \pmod p. \tag{9}$$

After the lists f_i, g_j are computed, we can collect the product

$$A = \prod_{\substack{id \pm j = q \\ B_1 < q \le B_2}} (f_i - g_j) \bmod N. \tag{10}$$

If there is a prime q in $]B_1, B_2]$ such that (8) holds, the product (10) will include i, j such that (9) holds, and thus $p \mid \gcd(A, N)$.

Stage 1 of P–1 computes $x_1 = x_0^e \pmod N$ and we can set $X_1 = x_1 + 1/x_1$ to make the P–1 stage 1 output compatible with our stage 2 at the cost of one modular inverse. Stage 1 of P+1 computes $x_1 = V_e(x_0) = V_e(\alpha_0 + 1/\alpha_0) = \alpha_0^e + 1/\alpha_0^e$ and we may simply set $X_1 = x_1$.

For P–1 stage 2, we could also use $f_i = x_1^{id} \bmod N$ and $g_j = x_1^j \bmod N$, for $1 \le j < d$ and $j \perp d$, instead of (6). An advantage of using (6) is that

$V_{-n}(X) = V_n(X)$, so that $g_j = V_j(X) \bmod N$ needs to be computed only for $1 \le j < d/2$, and one (i, j)-pair can sometimes include two primes at once. The same could be achieved by using $f_i = x_1^{(id)^2}$ and $g_j = x_1^{j^2}$, but computing these values for successive i or j via $(x^{(n+1)^2}, x^{2(n+1)+1}) = (x^{n^2} \cdot x^{2n+1}, x^{2n+1} \cdot x^2)$ costs two multiplications, whereas $V_{n+1}(x) = V_n(x)V_1(x) - V_{n-1}(x)$ costs only one. However, a modular inverse is required to convert the P–1 stage 1 output into the required form. Which approach is better thus depends on the choice of stage 2 parameters, i.e., on how many values need to be precomputed for the f_i and g_j lists. Assuming a small B_1, when using $B2 \approx 5000$ and $d = 210$, we need about 24 values for f_i and another 24 for g_j. The cost of a modular inverse is roughly 50 times the cost of a modular multiplication in our implementation, so the two approaches are about equally fast. Using the same stage 2 for P–1 and P+1 has the advantage of requiring only one implementation for both methods.

For ECM, we again would like two lists f_i and g_j such that $f_i \equiv g_j \pmod p$ if $id \cdot P_1 = j \cdot P_1$ on E_p, where P_1 is the point that was output by ECM stage 1. We can use $f_i = (id \cdot P_1)_X$, the X-coordinate of $id \cdot P_1$, and $g_j = (jP_1)_X$. A point and its inverse have the same X-coordinate on curves in Weierstraß and Montgomery form, so again we have $f_i - g_j \equiv 0 \pmod p$ if $q \mid id \pm j$. With points in projective coordinates, the points need to be canonicalized first to ensure that identical points have identical X-coordinates, which is described in Section 7.2.

How to choose parameter d and the set (i, j)-pairs needed during stage 2 for given B_1 and B_2 values is described in Section 7.1. Section 7.2 shows how to compute the lists f_i and g_j efficiently, and Section 7.3 describes how to accumulate the product (10).

7.1 Generating plans

The choice of d, the sets of i and j values to use for generating f_i and g_j, respectively, and the set of (i, j)-pairs for which to accumulate the product of $f_i - g_j$ depend on the B_1 and B_2 parameters for stage 2, but are independent of N, the number to factor. These choices are precomputed for given B_1 and B_2 and are stored as a "stage 2 plan." The stage 2 implementation then carries out the operations described by the plan, using arithmetic modulo N.

The plan provides the values d, i_0, i_1, a set S and a set T, chosen so that all primes q in $]B_1, B_2]$ appear as $q = id \pm j$ for some $(i, j) \in T$ with $i_0 \le i \le i_1$ and $j \in S$.

We try to choose parameters that minimize the number of group operations required for building the lists f_i and g_j and minimize the number of (i, j)-pairs required to cover all primes in the $]B_1, B_2]$ interval. This means that we would like to maximise i_0, minimize i_1, and cover two primes in $]B_1, B_2]$ with a single (i, j)-pair wherever possible.

We choose d highly composite and $S = \{1 \le j < d/2, j \perp d\}$, so that all integers coprime to d, in particular all primes not dividing d, can be written as $id \pm j$ for some integer i and $j \in S$. We assume B_1 is large enough that no prime greater than B_1 divides d. Choosing values of $i_0 = \lfloor B_1/d \rceil$ and $i_1 = \lfloor B_2/d \rceil$ is sufficient, but may be improved as shown below.

Computing the lists f_i and g_j requires at least one group operation per list entry, which is expensive especially in the case of ECM. The list f_i has $i_1 - i_0 + 1$

entries where $i_1 - i_0 \approx (B_2 - B_1)/d$, and g_j has $\phi(d)/2$ entries, so we choose d highly composite to achieve small $\phi(d)$ and try to minimize $i_1 - i_0 + 1 + \phi(d)/2$ by ensuring that $i_1 - i_0 + 1$ and $\phi(d)/2$ are about equally large. In our application of finding primes up to, say, 2^{32} as limited by the large prime bound used in the NFS sieving step, the value of B_2 will usually be of the order of a few thousand, and a choice $d = 210$ works well in this case. With $B_2 = 5000$, $i_1 = 24$ and $|S| = 24$, so the two lists of f_i and g_j are about equally large, assuming small i_0. For smaller B_2, a smaller d is preferable, for example $d = 90$ for $B_1 = 100$, $B_2 = 1000$.

We have chosen i_1 as an upper bound based on B_2, but we may reduce i_1 yet if $[i_1 d - d/2, i_1 d + d/2]$ does not include any primes up to B_2, and so obtain the final value of i_1.

Having chosen d, S, and i_1, we can choose T. We say a prime $q \in]B_1, B_2]$ is covered by an (i, j)-pair if $q \mid id \pm j$; assuming that only the largest prime factor of any $id \pm j$ value lies in $]B_1, B_2]$, each pair may cover up to two primes. For each prime $q \in]B_1, B_2]$ we mark the corresponding entry $a[q]$ in an array to signify a prime that yet needs to be covered.

Let r be the smallest prime not dividing d. Then $q \mid id \pm j$ and $q \neq id \pm j$ implies $q = (id \pm j)/s$ with $s \geq r$ since $id \pm j \perp d$, thus $q \leq (id \pm j)/r$. Hence composite values of $id \pm j$ with $i \leq i_1$ can cover only primes up to $\lfloor (i_1 d + d/2)/r \rfloor$, and each prime $q > \lfloor (i_1 d + d/2)/r \rfloor$ can be covered only by $q = id \pm j$.

In a first pass, we examine each prime q, $\lfloor (i_1 d + d/2)/r \rfloor < q \leq B_2$, highest to lowest and the (i, j)-pair covering this prime. This pair is the only way to cover q and must eventually be included in T. If this (i, j)-pair also covers a smaller prime q' as a composite value, then $a[q']$ is un-marked.

In a second pass, we look for additional (i, j)-pairs that cover two primes, both via composite values. We examine each (i, j)-pair with $i_0 \leq i \leq i_1$ highest to lowest, and $j \in S$. If there are two primes q' and q'' marked in the array that are covered by the (i, j)-pair under examination, then $a[q']$ and $a[q'']$ are un-marked, and $a[id - j]$ is marked instead.

In the third pass, we cover the remaining primes $q \leq \lfloor (i_1 d + d/2)/r \rfloor$ using (i, j)-pairs with large i, if possible, hoping that we may increase the final i_0 value. As in the second pass, we examine each (i, j)-pair in order of decreasing i and, if there is a prime q' with $a[q']$ marked, $q' \mid id \pm j$ but $q' \neq id \pm j$, we un-mark $a[q']$ and mark $a[id - j]$ instead. This way, all primes in $]B_1, B_2]$ are covered, and each with an (i, j)-pair with the largest possible $i \leq i_1$.

We now choose the final i_0 value by looking for the smallest (not necessarily prime) q such that a $a[q]$ is marked, and setting $i_0 = \lfloor q/d \rfloor$. The set T is determined by including each (i, j)-pair where an array element $a[id - j]$ or $a[id + j]$ is marked. The pairs in T are stored in order of increasing i so that the f_i can be computed sequentially for P–1 and P+1.

7.2 Initialisation

In the initialisation phase of stage 2 for P–1 and P+1 (and similarly for ECM), we compute the values $g_j = V_j(X_1)$ with $1 \leq j < d/2$, $j \perp d$ and set up the computation of $f_i = V_{id}(X_1)$ for $i_0 \leq i \leq i_1$. To do so, we need Lucas chains that generate all required values of id and j. We try to find a short Lucas chain that produces all required values to save group operations which are costly especially for ECM.

Lucas chains for values in an arithmetic progression are particularly simple, since the difference of successive terms is constant. We merely need to start the chain with terms that generate the common difference and the first two terms of the arithmetic progression.

The values of j with $j \perp d$ and $6 \mid d$ can be computed in two arithmetic progressions $1+6m$ and $5+6m$, via the Lucas chain $1, 2, 3, 5, 6, 7, 11, 13, 17, 19, 23, \ldots$ For $d = 210$, the required 24 values of j can therefore be generated with a Lucas chain of length 37.

To add the values of id with $i_0 \leq i \leq i_1$, we need to add d, $i_0 d$, and $(i_0+1)d$ to the chain. If $2 \mid\mid d$, we have $d/2-2 \perp d$ and $d/2+2 \perp d$ and we can add d to the Lucas chain by including $4 = 2+2$ and $d = d/2+2+d/2-2$. If $4 \mid d$, we have $d/2-1 \perp d$ and $d/2+1 \perp d$ and we can add d simply via $d = d/2+1+d/2+1$ as 2 is already in the chain. Since i_0 is usually small, we can compute both $i_0 d$ and $(i_0+1)d$ from d with one binary chain.

Using this Lucas chain, we can compute and store all the $g_j = V_j(X_1)$ residues as well as $V_d(X_1)$, $f_{i_0 d}(X_1)$, and $f_{(i_0+1)d}(X_1)$.

In the case of P–1 and P+1, since the (i, j)-pairs are stored in order of increasing i, all the f_i values need not be computed in advance, but can be computed successively as the (i, j)-pairs are processed.

7.2.1 Initialisation for ECM

For ECM stage 2 we use curves in Montgomery form with projective coordinates, just as in stage 1, to avoid costly modular inverses. The initialisation uses the same Lucas chain as in 7.2 for the required values of id and j, so that $id \cdot P_1$ and $j \cdot P_1$ can be computed efficiently. However, two points $(X_1 :: Z_1)$ and $(X_2 :: Z_2)$ in projective coordinates being identical does not imply $X_1 = X_2$, but $X_1/Z_1 = X_2/Z_2$, where Z_1 and Z_2 are generally not equal, so the X-coordinates of these points cannot be used directly to build the lists f_i and g_j.

There are several ways to overcome this obstruction. Several authors (for example [7, 7.4.2] or [24]) propose storing both X and Z coordinate in the f_i and g_j lists, then accumulating the product $A = \prod_{(i,j)\in T} ((f_i)_X (g_j)_Z - (g_j)_X (f_i)_Z)$. An advantage of this approach is that the f_i can be computed sequentially while the product is being accumulated and the number of g_j to precompute and store can be controlled by choice of d, which allows ECM stage 2 to run under extremely tight memory conditions such as in an FPGA implementation. The obvious disadvantage is that each (i, j)-pair now uses 3 modular multiplications instead of 1 as in (10).

Another approach and much preferable in an implementation where sufficient memory is available is canonicalizing the precomputed points so that all points have the same Z-coordinate. To produce the desired lists f_i and g_j, we therefore compute all the required points $Q_i = id \cdot P_1$ and $R_j = j \cdot P_1$. If all Z-coordinates of Q_i and R_j are made identical, $Q_i = R_j$ on E_p implies $(Q_i)_X \equiv (R_j)_X \pmod{p}$, as desired, and we may set $f_i = (Q_i)_X$ and $g_j = (R_j)_X$.

We suggest two methods for this. One is to set all Z-coordinates to 1 $\pmod{N}$ via $(X :: Z) = (XZ^{-1} :: 1)$. To do so, we need the inverse modulo N of each Z-coordinate of our precomputed points. A trick due to Montgomery, described for example in [6, 10.3.4], replaces n modular inverses of residues r_k modulo N, $1 \leq k \leq n$, by 1 modular inverse and $3n-3$ modular multiplications. This way we can canonicalize a batch of n points with $4n-3$ modular

multiplications and 1 modular inverse. Not all points needed for the f_i and g_j lists need to be processed in a single batch; if memory is insufficient, the points needed for f_i can be processed in several batches while product (10) is being accumulated.

A faster method was suggested by P. Zimmermann. Given $n \geq 2$ points $P_1, \ldots, P_n$, $P_i = (X_i :: Z_i)$, we set all Z-coordinates to $\prod_{1 \leq i \leq n} Z_i$ by multiplying each X_k by $T_k = \prod_{1 \leq i \leq n, i \neq k} Z_i$. This can be done efficiently by computing two lists $s_k = \prod_{1 \leq i \leq k} Z_i$ and $t_k = \prod_{k < i \leq n} Z_i$ for $1 \leq k < n$, each at the cost of $n-2$ modular multiplications. Now we can set $T_1 = t_1, T_n = s_{n-1}$, and $T_i = s_{i-1} t_i$ for $1 < i < n$, also at the cost of $n-2$ multiplications. Multiplying X_i by T_i costs another n modular multiplications for a total of only $4n-6$ modular multiplications, without any modular inversion. Algorithm 4 implements this idea. Since the common Z-coordinate of the canonicalized points is the product of all points idP_1 and jP_1, the complete set of points needed for the f_i and g_j lists must be processed in a single batch.

Interestingly, if the curve parameters are chosen such that the curve initialisation can be done with modular division by small constants rather than with a modular inverse, then ECM implemented this way does not use any modular inverses at all, without sacrificing the optimal cost of 1 modular multiplication per (i, j)-pair in stage 2.

Input: $n \geq 2$, an integer
  N, a positive integer
  $Z_1, \ldots, Z_n$, residues modulo N
Data: s, a residue modulo N
Output: $T_1, \ldots, T_n$, residues modulo N with $T_i \equiv \prod_{1 \leq i \leq n, i \neq k} Z_i \pmod N$
$T_{n-1} := Z_n$;
for $k := n-1$ **downto** 2 **do**
  $T_{k-1} := T_k \cdot Z_k \bmod N$;
$s := Z_1$;
$T_2 := T_2 \cdot s \bmod N$;
for $k := 3$ **to** n **do**
  $s := s \cdot Z_{k-1} \bmod N$;
  $T_k := T_k \cdot s \bmod N$;

Algorithm 4: Batch cross multiplication algorithm.

7.3 Executing plans

The stage 2 plan stores the (i, j)-pairs which cover all primes in $]B1, B2]$. The f_i and g_j lists are computed as described in 7.2. Stage 2 then reads the stored (i, j)-pairs, and multiplies $f_i - g_j$ to an accumulator:

$$A = \prod_{(i,j) \in T} (f_i - g_j) \bmod N. \tag{11}$$

Since the pairs are stored in order of increasing i, the full list f_i need not be precomputed for P–1 and P+1, but each f_i can be computed sequentially by $V_{(i+1)d}(X_1) = V_{id}(X_1) V_d(X_1) - V_{(i-1)d}(X_1)$. At the end of stage 2, we take $r = \gcd(A, N)$, hoping that $1 \leq r \leq N$ and so that r is a proper factor of N.

7.3.1 Backtracking

We would like to avoid finding all prime factors of the input number N simultaneously, i.e., finding N as a trivial factor. As in stage 1, a backtracking mechanism is used to recover from such a situation.

Since $r = \gcd(A, N)$ and A is a reduced residue modulo N, we find $r = N$ as a factor if and only if $A = 0$. We set up a "backup" residue $A' = 1$ at the start of evaluation of (11). At periodic intervals during the evaluation of (11), for example each time that i is increased, we test if $A = 0$, which is fast since the residue does not need to be converted out of Montgomery representation if REDC (see Section 3.2) is used for the arithmetic. If $A = 0$, we take $r = \gcd(A', N)$ and end stage 2. Otherwise, we set $A' = A$. This way, a proper factor of N can be discovered so long as all prime factors of N are not found between two tests for $A = 0$.

7.4 P+1 and ECM stage 2 performance

Tables 6 and 7 compares the performance of the P+1 and the ECM stage 2 implementation for different B_2 values and modulus sizes on AMD Phenom and Intel Core 2 CPUs. In each case, the timing run used $B_1 = 10$ and $d = 210$, and the time for a run with $B_1 = 10$ and without any stage 2 was subtracted.

	Core 2		Phenom	
B_2	1 word	2 words −2 bits	1 word	2 words −2 bits
1000	3.06	6.72	2.91	6.24
2000	4.09	9.86	3.64	8.08
3000	5.07	12.7	4.37	10.1
4000	6.00	15.5	5.01	11.8
5000	6.95	18.2	5.77	13.8
6000	7.80	20.8	6.40	15.4
7000	8.83	23.7	7.09	17.3
8000	9.69	26.3	7.73	19.0
9000	10.7	29.0	8.39	20.7
10000	11.5	31.4	9.01	22.5
20000	20.3	57.0	15.3	39.3
30000	28.9	81.8	21.3	55.0
40000	37.4	106	27.2	70.8
50000	45.7	130	33.1	86.2
60000	54.1	154	38.8	102

Table 6: Time in microseconds for P+1 stage 2 with different B_2 values on 2.146 GHz Intel Core 2 and 2 GHz AMD Phenom CPUs

7.5 Overall performance of P−1, P+1 and ECM

Tables 8 and 9 shows the expected time to find primes close to $2^{25}, 2^{26}, \ldots, 2^{32}$ for moduli of 1 word and of 2 words, and the B_1 and B_2 parameters chosen to minimize the expected time. The probability estimate is based on the elliptic curve with rational 12 torsion and parameter $k = 2$ in 6.2. That the B_1 and B_2 parameters are not monotonously increasing with factor size is due to the

	Core 2		Phenom	
B_2	1 word	2 words −2 bits	1 word	2 words −2 bits
1000	5.86	17.2	7.10	17.5
2000	7.46	21.5	7.87	19.7
3000	8.83	25.4	8.79	22.0
4000	10.1	29.7	9.55	24.1
5000	11.5	33.7	10.5	26.5
6000	12.7	37.6	11.2	28.2
7000	14.0	41.4	12.1	30.8
8000	15.4	45.8	12.9	32.7
9000	16.7	49.6	13.7	34.6
10000	17.9	53.4	14.5	36.9
20000	30.5	91.3	22.3	56.6
30000	42.8	128	29.7	75.0
40000	54.9	164	37.2	94.3
50000	66.7	200	44.5	113
60000	78.3	235	51.8	131

Table 7: Time in microseconds for ECM stage 2 with different B_2 values on 2.146 GHz Intel Core 2 and 2 GHz AMD Phenom CPUs

fact that the expected time to find a prime factor as a function of B_1 and B_2 is very flat around the minimum, so that even small perturbations of the timings noticeably affect the parameters chosen as optimal.

n	B_1	B_2	Prob.	1 word	2 words −2 bits
25	300	5000	0.249	46	103
26	310	6000	0.220	55	125
27	320	6000	0.186	67	151
28	400	6000	0.167	81	182
29	430	7000	0.149	100	224
30	530	11000	0.158	119	275
31	530	10000	0.128	144	330
32	540	10000	0.105	177	

Table 8: Expected time in microseconds and probability to find prime factors close to 2^n of composites with 1 or 2 words with P–1 on 2 GHz AMD Phenom CPUs. The B_1 and B_2 parameters are chosen to minimize the time/probability ratio.

8 Comparison to hardware implementations of ECM

Several hardware implementations of ECM for use as a cofactorization device in NFS have been described recently, based on the proposed design "SHARK" by Franke et al. [11] SHARK is a hardware implementation of GNFS for factoring 1024-bit integers which uses ECM for cofactorization after sieving. The idea of implementing GNFS in hardware is inspired by the observation of Bernstein [1]

n	B_1	B_2	Prob.	1 word	2 words −2 bits
25	130	7000	0.359	67	176
26	130	7000	0.297	81	213
27	150	11000	0.290	101	264
28	160	13000	0.256	124	324
29	180	12000	0.220	151	395
30	200	12000	0.188	190	496
31	260	14000	0.182	231	604
32	250	15000	0.147	283	744

Table 9: Expected time in microseconds and probability per curve to find prime factors close to 2^n of composites with 1 or 2 words with ECM on 2 GHz AMD Phenom CPUs. The B_1 and B_2 parameters are chosen empirically to minimize the expected time.

that dedicated hardware could achieve a significantly lower cost in terms of Area-Time product than a software implementation that uses sieving on a regular PC. He proposes, among other algorithms, to use ECM for the smoothness test.

Pelzl et al. [24] present a scalable implementation of ECM stage 1 and stage 2 for input numbers of up to 200 bits, based on Xilinx Virtex2000E-6 FPGAs with an external microcontroller. Their design has one modular multiplication unit per ECM unit, and each ECM unit performs both stage 1 and stage 2. They propose using the bounds $B_1 = 910$ and $B_2 = 57000$ for finding primes of up to about 40 bits. They use curves in Montgomery form (3) and a binary Lucas chain for stage 1 that uses 13740 modular multiplications (including squarings), and estimate that an optimized Lucas chain could do it in ≈ 12000 modular multiplications. They use an enhanced standard stage 2 that uses 3 modular multiplications per (i, j)-pair, see 7.2. With a value $d = 210$, they estimate 303 point additions and 14 point doublings in the initialisation of stage 2, and 13038 modular multiplications for collecting the product (10) with 4346 different (i, j)-pairs for a total of 14926 modular multiplications in stage 2. However, to minimize the AT product, they propose using $d = 30$ with a total of 24926 modular multiplications in stage 2.

In our implementation, stage 1 with $B_1 = 910$ and PRAC-generated chains (using cost 6 for point addition, 5 for doubling, 0.5 for each byte code and 0.5 for each byte code change as parameters for rating candidate chains) uses 11403 modular multiplications, 83% of the figure for the binary Lucas chain. (Using chains for composite values where the resulting chain is shorter than the concatenated chains for the factors is not currently used and could probably reduce this figure by a few more percent.) Our stage 2 with $B_2 = 57000$ and $d = 210$ uses 290 point additions, 13 point doublings, 1078 modular multiplications for point canonicalization and 4101 pairs which cost 1 modular multiplication each, for a total of 6945 modular multiplications. The cost of computing and canonicalizing the points idP_1 has a relatively large share in this figure, suggesting that a value for d such that $B_2/(d\phi(d))$ is closer to 1 might reduce the total multiplication count. In a hardware implementation, the extra memory requirements may make larger d values inefficient in terms of the AT product, but this is not an issue in a software implementation on a normal PC. In our implementation, $d = 630$ provides the minimum total number of 5937 modular multiplications in

stage 2, only 40% of the number reported by Pelzl et al. for $d = 210$, and only 24% of their number for $d = 30$.

These figures suggest that a software implementation of ECM on a normal PC enjoys an advantage over an implementation in embedded hardware by having sufficient memory available that choice of algorithms and of parameters are not constrained by memory, which significantly reduces the number of modular multiplications in stage 2. This problem might be reduced by separating the implementation of stage 1 and stage 2 in hardware, so that each stage 1 units needs only very little memory and forwards its output to a stage 2 unit which has enough memory to compute stage 2 with a small multiplication count, while the stage 1 unit processes the next input number.

Gaj et al. [12] improve on the design by Pelzl et al. mainly by use of a more efficient implementation of modular multiplication, by avoiding limitations due to the on-chip block RAM which allows them to fit more ECM units per FPGA, and removing the need for an external microcontroller. The algorithm of ECM stage 1 and stage 2 is essentially the same as that of Pelzl et al. They report an optimal performance/cost ratio of 311 ECM runs per second per \$100 for an input number of up to 198 bits with $B_1 = 910, B_2 = 57000, d = 210$, using an inexpensive Spartan 3E XC3S1600E-5 FPGA for their implementation. They also compare their implementation to an ECM implementation in software, GMP-ECM [10], running on a Pentium 4, and conclude that their design on a low-cost Spartan 3 FPGA offers about 10 times better performance/cost ratio than GMP-ECM on a Pentium 4. However, GMP-ECM is a poor candidate for assessing the performance of ECM in software for very small numbers with low B_1 and B_2 values. GMP-ECM is optimized for searching large prime factors (as large as reasonably possible with ECM) of numbers of at least a hundred digits size by use of asymptotically fast algorithms in particular in stage 2, see [29]. For very small input, the function call and loop overhead in modular arithmetic and the cost of generating Lucas chains on-the-fly in stage 1 dominates the execution time; likewise in stage 2, the initialisation of the polynomial multipoint evaluation and again function call and loop overhead will dominate, while the B_2 value is far too small to let the asymptotically fast stage 2 (with time in $\tilde{O}(\sqrt{B_2})$) make up for the overhead.

De Meulenaer et. al. [8] further improve the performance/cost-ratio by using a high-performance Xilinx Virtex4SX FPGA with embedded multipliers instead of implementing the modular multiplication with general-purpose logic. They implement only stage 1 of ECM and only for input of up to 135 bits. One ECM unit utilizes all multipliers of the selected FPGA, so one ECM unit fits per device. By scaling the throughput of the design of Gaj et al. to 135-bit input, they conclude that their design offers a 15.6 times better performance/cost ratio. In particular, assuming a cost of \$116 per device, they state a throughput of 13793 ECM stage 1 with $B_1 = 910$ per second per \$100.

We compare the cost of finding 40-bit factors using our software implementation of ECM with that given by de Meulenaer et al. Our implementation is currently limited to moduli of size 2 words with the two most significant bits zero, or 126 bits on a 64-bit system, whereas the implementation of de Meulenaer et al. allows 135-bit moduli. Extending our implementation to numbers of 3 words is in progress, but not functional at this time. We expect that ECM with 3-word moduli will take about twice as long as for 2-word moduli. For

Device	XC4VSX25-10	Phenom 9350e	Phenom II X4 955
Clock rate	0.22 GHz	2.0 GHz	3.2 GHz
Cores per device	1	4	4
126-bit modulus (2 words in software)			
Time per stage 1	$62.5\mu s$	$232.1\mu s$	$\approx 145\mu s$
Time per stage 2	$59.2\mu s$	$121.5\mu s$	$\approx 76\mu s$
Time per trial	121.7	$353.6\mu s$	$\approx 221\mu s$
#Trials/sec/device	8217	11312	18100
Cost per device	\$300		\$215
#Trials/sec/\$100	2739		8418
135-bit modulus (3 words in software)			
Time per stage 1	$62.5\mu s$	$\approx 464\mu s$	$\approx 290\mu s$
Time per stage 2	$59.2\mu s$	$\approx 243\mu s$	$\approx 152\mu s$
Time per trial	121.7	$\approx 707\mu s$	$\approx 442\mu s$
#Trials/sec/device	8217	≈ 5658	≈ 9052
Cost per device	\$300		\$215
#Trials/sec/\$100	2739		4210

Table 10: Comparison of ECM with $B_1 = 910$, $B_2 = 57000$ for 126-bit and 135-bit input on a Virtex4SX25-10 FPGA and on AMD 64-bit microprocessors.

the comparison we use timings for 126-bit moduli (2 words) and estimates for 135-bit moduli (3 words).

The timings for our code are obtained using an AMD Phenom X4 9350e with four cores at 2.0 GHz. The AMD 64-bit CPUs all can perform a full 64×64-bit product every 2 clock cycles, making them an excellent platform for multi-precision modular arithmetic. The fastest AMD CPU currently available is a four-core 3.2 GHz Phenom II X4 955 at a cost of around \$215 (regular retail price, according to `www.newegg.com` on July 28th 2009) and we scale the timings linearly to that clock rate. Since the code uses almost no resources outside the CPU core, linear scaling is reasonable. The number of clock cycles used is assumed identical between the Phenom and Phenom II. Similarly, running the code on n cores of a CPU is assumed to increase total throughput n-fold.

Table 10 compares the performance of the implementation in hardware of de Meulenaer et al. and of our software implementation, using the parameters $B_1 = 910$, $B_2 = 57000$. The software implementation uses $d = 630$ for stage 2. De Meulenaer et al. do not implement stage 2, but predict its performance as capable of $16,900$ stage 2 per second per device. We use this estimate in the comparison. They also give the cost of one Xilinx XC4VSX25-10 FPGA as \$116 when buying 2500 devices. The current quote at `www.nuhorizons.com` and `www.avnet.com` for this device is about \$300, however. We base the price comparison on the latter figure. Only the cost of the FPGA or the CPU, respectively, are considered. The results show that a software implementation of ECM can compete in terms of cost per ECM trial with the published designs for ECM in hardware. An advantage of the software implementation is flexibility: it can run on virtually any 64-bit PC, and so utilize otherwise idle computing resources. If new systems are purchased, they involve only standard parts that can be readily used for a wide range of computational tasks. Given a comparable

performance/cost ratio, an implementation in software running on standard hardware is the more practical.

Our current implementation is sufficient for one set of parameters proposed by the SHARK [11] design for factoring 1024-bit integers by GNFS which involves the factorization of approximately $1.7 \cdot 10^{14}$ integers of up to 125 bits produced by the sieving step. The time for both stage 1 and stage 2 with $B_1 = 910, B_2 = 57000$ is $353.6\mu s$ on a 2 GHz Phenom, and about $221\mu s$ on a 3.2 GHz Phenom II. Using the latter, $1.7 \cdot 10^{14}$ ECM trials can be performed in approximately 300 CPU-years. But how many curves need to be run per input number? Pelzl et al. [24] state that 20 curves at $B_1 = 910$, $B_2 = 57000$ find a 40-bit factor with $> 80\%$ probability, and doing 20 trials per input number gives a total time of about 6000 CPU years. However, the vast majority of input numbers will not be 2^{40}-smooth, and fewer than 20 curves suffice to establish non-smoothness with high probability, making this estimate somewhat pessimistic. Assuming a cost of about \$350 for a bare-bone but functional system with one AMD Phenom II X4 955 CPU, this translates to a pessimistic estimate of about \$2.1M for hardware capable of performing the required ECM factorizations within a year.

References

[1] Daniel J. Bernstein. Circuits for integer factorization: a proposal. Manuscript, 2001. `http://cr.yp.to/nfscircuit.html`.

[2] Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980.

[3] Richard P. Brent, Richard E. Crandall, Karl Dilcher, and Christopher van Halewyn. Three new factors of Fermat numbers. *Mathematics of Computation*, 69(231):1297–1304, 2000.

[4] Stefania Cavallar. Three-Large-Primes Variant of the Number Field Sieve. Technical Report MAS-R0219, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, August 2002.

[5] Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Walter Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, Paul Leyland, Jöel Marchand, François Morain, Alec Muffett, Chris and Craig Putnam, and Paul Zimmermann. Factorization of a 512-Bit RSA Modulus. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2000.

[6] Henri Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag, 1st edition, 1995.

[7] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective.* Springer-Verlag, 2nd edition, 2005.

[8] Giacomo de Meulenaer, François Gosset, Guerric Meurice de Dormale, and Jean-Jacques Quisquater. Elliptic Curve Factorization Method : Towards Better Exploitation of Reconfigurable Hardware. In *IEEE Symposium on*

Field-Programmable Custom Computing Machines (FCCM07), pages 197–207. IEEE Computer Society Press, 2007.

[9] Reina-Marije Elkenbracht-Huizing. *Factoring integers with the Number Field Sieve*. PhD thesis, Rijksuniversiteit te Leiden, 1997.

[10] Paul Zimmermann et al. The ECMNET Project. `http://www.loria.fr/~zimmerma/records/ecmnet.html`.

[11] Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, and Colin Stahlke. SHARK: A Realizable Special Hardware Sieving Device for Factoring 1024-Bit Integers. In *Cryptographic Hardware and Embedded Systems — CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 119–130, 2005.

[12] Kris Gaj, Soonhak Kwon, Patrick Baier, Paul Kohlbrenner, Hoang Le, Mohammed Khaleeluddin, and Ramakrishna Bachimanchi. Implementing the Elliptic Curve Method of Factoring in Reconfigueable Hardware. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 2006.

[13] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 61–72, New York, NY, USA, 1994. ACM.

[14] Thorsten Kleinjung. Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024 bit integers. In *Proceedings of Special-purpose Hardware for Attacking Cryptographic Systems 2006 (SHARCS'06)*.

[15] Arjen K. Lenstra and Hendrik W. Lenstra, Jr., editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1993.

[16] Hendrik W. Lenstra, Jr. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[17] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985.

[18] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48:243–264, 1987.

[19] Peter L. Montgomery. Evaluating Recurrences of Form $X_{m+n} = f(X_m, X_n, X_{m-n})$ Via Lucas Chains. Unpublished Manuscript, 1992. `ftp://ftp.cwi.nl/pub/pmontgom/Lucas.ps.gz`.

[20] Peter L. Montgomery. *An FFT Extension to the Elliptic Curve Method of Factorization*. PhD thesis, UCLA, 1992. `ftp://ftp.cwi.nl/pub/pmontgom/ucladissertation.psl.gz`.

[21] Peter L. Montgomery. Personal Communication, 2001.

[22] Peter L. Montgomery. Personal Communication, 2008.

[23] Peter L. Montgomery and Alexander Kruppa. Improved Stage 2 to $p \pm 1$ Factoring Algorithms. In Alfred J. van der Poorten and Andreas Stein, editors, *Proceedings of the 8th Algorithmic Number Theory Symposium (ANTS VIII)*, volume 5011 of *Lecture Notes in Computer Science*, pages 180–195. Springer-Verlag, 2008.

[24] Jan Pelzl, Martin Šimka, Thorsten Kleinjung, Jens Franke, Christine Priplata, Colin Stahlke, Miloš Drutarovský, Viktor Fischer, and Christof Paar. Area-Time Efficient Hardware Architecture for Factoring Integers with the Elliptic Curve Method. *IEE Proceedings Information Security*, 152(1):67–78, 2005.

[25] John M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[26] Joseph J. Silverman and John Tate. *Rational Points on Elliptic Curves*. Springer-Verlag, 1995.

[27] Hiromi Suyama. Informal preliminary report (8). Letter to Richard P. Brent, October 1985.

[28] George Woltman and Scott Kurowski. The Great Internet Mersenne Prime Search. `http://www.gimps.org/`.

[29] Paul Zimmermann and Bruce Dodson. 20 Years of ECM. In Florian Hess, Sebastian Pauli, and Michael Pohst, editors, *Proceedings of the 7th Algorithmic Number Theory Symposium (ANTS VII)*, volume 4076 of *Lecture Notes in Computer Science*, pages 525–542. Springer-Verlag, 2006.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399