Faster cofactorization with ECM using mixed representations

Cyril Bouvier and Laurent Imbert

LIRMM, CNRS, Univ. Montpellier, France

Abstract. This paper introduces a novel implementation of the elliptic curve factoring method specifically designed for medium-size integers such as those arising by billions in the cofactorization step of the Number Field Sieve. In this context, our algorithm requires fewer modular multiplications than any other publicly available implementation. The main ingredients are: the use of batches of primes, fast point tripling, optimal double-base decompositions and Lucas chains, and a good mix of Edwards and Montgomery representations.

Keywords: Elliptic curve method, cofactorization, double-base representation, twisted Edwards curve, Montgomery curve, CADO-NFS

1 Introduction

The Elliptic Curve Method (ECM) invented by H. W. Lenstra Jr. in 1985 [18] is probably the most versatile algorithm for integer factorization. It remains the asymptotically fastest known method for finding medium-size prime factors of large integers. The 50 largest factors found with ECM have 68 to 83 digits; they are recorded in [26]. ECM is also a core ingredient of the Number Field Sieve (NFS) [17], the most efficient general purpose algorithm for factoring "hard" composite integers of the form $N = pq$ with $p, q \approx \sqrt{N}$. ECM is equally essential in all NFS variants for computing discrete logarithms over finite fields. In NFS and its variants, ECM is used as a subroutine of the sieving phase. It is also employed in the descent phase for discrete logarithms computations. Together with other factoring algorithms such as the Quadratic Sieve, $p - 1$ or $p + 1$, it is extensively used in the so-called cofactorization step. This important step consists of breaking into primes billions of composite integers of a hundred-ish bits that are known to have no small prime factor. The time spent in ECM for these medium-size, yet hard to factor integers is therefore substantial. For example, with CADO-NFS [24], the cofactorization time for a 200-digit RSA number represents between 15 % and 22 % of the sieving phase. According to [9], cofactorization represented roughly one third of the sieving phase and 5% to 20% of the total wall-clock time in the current world-record factorization of a 768-bit RSA number [16]. For larger factorization or discrete logarithm computations, Bos and Kleinjung anticipate that the time spent in cofactorization, notably ECM, becomes more and more important [9].

Since its invention, ECM has been the subject of many improvements [27]. It has been shown that the choice of "good" elliptic curve representations and parameters plays an important role in both the efficiency of ECM and its probability of success. Historically, Lenstra considered short Weierstrass curves together with Jacobian coordinates. Then, Montgomery introduced a new model for elliptic curves together with a system of coordinates perfectly suited to ECM [22]. Montgomery curves have been the best option for about twenty five years. This setting is used in GMP-ECM [28], a state-of-the-art implementation of ECM. More than twenty five years later, building over the works of Edwards [13], Bernstein et al. proposed an efficient implementation of ECM using twisted Edwards curves [4]. Yet, there is no clear general answer to the question of which curve model is best suited to ECM.

In this work, we propose an algorithm specifically designed for the medium-size integers that occur in the cofactorization step of NFS. We extend ideas from Dixon and Lenstra [12] and from Bos and Kleinjung [9] by processing the scalar of the first stage of ECM by batches of primes. Unlike [9] and [15] which only consider NAF decompositions for these batches, we take advantage of the fastest known tripling formula on twisted Edwards curves [6] together with optimal double-base decompositions. Finally, we also use Lucas chains by exploiting the birational equivalence between twisted Edwards curves and Montgomery curves and by switching from one model to the other when appropriate. Our algorithm performs fewer modular multiplications than any other publicly available implementation. Our results are implemented in the CADO-NFS software [24]. Updates and more detailed data will be posted online at `http://eco.lirmm.net/double-base_ECM/`.

2 Preliminaries

In this section, we present the basics of ECM. Then we recall the definitions of Montgomery curves and twisted Edwards curves together with the associated point representations and arithmetic operations.

In order to compare the cost of the different elliptic operations and scalar multiplication algorithms, we count the number of modular multiplications ($\mathbf{M}$) and squarings ($\mathbf{S}$). To ease the comparisons, we assume that both operations take the same time (i.e. $1\mathbf{S} = 1\mathbf{M}$) as in other ECM implementation papers [4,9,15].[1]

2.1 The Elliptic Curve Method

Lenstra's elliptic curve method [18] is often viewed as a generalization of Pollard's $p-1$ algorithm in the sense that it exploits the possible smoothness of the order of an elliptic curve defined over an unknown prime divisor of a given composite integer N.

[1] This claim is also supported by our experiments with CADO-NFS modular arithmetic functions for 64-bit, 96-bit and 128-bit integers.

ECM starts by choosing an elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$ and a point P on E. In practice, one usually selects a "random" curve $E_{\mathbb{Q}} : y^2 = x^3 + ax + b$ over $\mathbb{Q}$ together with a nontorsion point P' on $E_{\mathbb{Q}}$, and then reduces the curve parameters a, b and the coordinates of P' modulo N to get E and P. Unlike elliptic curves defined over finite fields, the set of points $E(\mathbb{Z}/N\mathbb{Z})$ contains non-affine points that are different from the point at infinity, i.e., projective points $(X : Y : Z)$ with $Z \neq 0$ and Z not invertible modulo N. For these "special" points, $\gcd(N, Z)$ gives a factor of N. The purpose of ECM is thus to produce these "special" points with a reasonably high probability and at reasonably low cost.

Let p be an unknown prime dividing N, and let E_p be the curve defined over $\mathbb{F}_p$ by reducing the equation of E modulo p. The goal of ECM is to produce (virtually) the point at infinity on E_p while carrying-out all the computations on E. ECM does so by computing $Q = [k]P \in E$ for a fixed scalar k. It achieves its goal whenever $\#E_p$ divides k. To that end, k is chosen such that, $\#E_p \mid k$ whenever $\#E_p$ is B_1-powersmooth for a carefully chosen bound B_1. (An integer is B-powersmooth if none of the prime powers dividing that integer is greater than B.) Most current implementations use $k = \mathrm{lcm}(2, 3, 4, \ldots, B_1)$ as it offers an excellent balance between efficiency and probability of success. For $B_1 \in \mathbb{N}$, we have:

$$k = \mathrm{lcm}(2, 3, 4, \ldots, B_1) = \prod_{p \text{ prime} \leq B_1} p^{\lfloor \log_p(B_1) \rfloor} \tag{1}$$

In the following, the multiset composed of all primes p less than or equal to B_1, each occurring exactly $\lfloor \log_p(B_1) \rfloor$ times, is denoted $\mathcal{M}_{B_1}$.

The approach described so far is often referred to as "stage 1". There is a "stage 2" continuation for ECM which takes as input an integer bound $B_2 \geq B_1$ and succeeds if the order $\#E_p$ is B_1-powersmooth except for one prime factor which may lie between B_1 and B_2.

In this article, we focus on the use of ECM as a subroutine of the NFS algorithm. In this case, the values of B_1 and B_2 are relatively small and usually hardcoded. For example, in CADO-NFS [24], the ECM computations are done with a predefined set of values for B_1 and B_2 (some possible values for B_1 are 105, 601 and 3517). In this context, it may be worthwhile to perform some precomputations on the hardcoded values.

2.2 Montgomery curves

Historically, the elliptic curve method was implemented using short Weierstrass curves. Montgomery curves were described in [22] to improve the efficiency of ECM by reducing the cost of elliptic operations. Montgomery curves are used in many implementations of ECM, for example in GMP-ECM [28], the most-widely used ECM implementation.

Definition 1 (Montgomery curve). *Let K be a field and $A, B \in K$ such that $B(A^2 - 4) \neq 0$. A Montgomery curve, denoted $E^M_{A,B}$, is an elliptic curve whose*

affine points are all $(x, y) \in K^2$ *such that*

$$By^2 = x^3 + Ax^2 + x. \tag{2}$$

In practice, projective coordinates $(X : Y : Z)$ are used to avoid field inversions. Montgomery proposed to drop the Y coordinate, performing the computations on X and Z only. In the so-called XZ coordinate system, a point is denoted $(X : : Z)$. An immediate consequence is that one cannot distinguish between a point and its opposite. This implies that, given two distinct points on the curve, one can compute their sum, only if one knows their difference. This new operation is called a *differential addition*.

As seen in Table 1, XZ coordinates on Montgomery curves allow for very fast point doubling and differential addition. However, the condition imposed by the use of a differential addition forces to use specific scalar multiplication algorithms (see Section 3.3).

Note that the doubling formula is often accounted for $2\mathbf{M} + 2\mathbf{S}$ plus one multiplication by a small constant. Yet, this operation count is relevant only when the curve coefficient A is chosen such that $(A+2)/4$ is small. In Table 1 we report a cost of $5\mathbf{M}$ for dDBL because our choice of parameterization prevents us from assigning any particular value to $(A + 2)/4$. We give more details in Section 2.5.

Table 1. Arithmetic cost of elliptic operations for Montgomery curves in XZ coordinates under the assumption $1\mathbf{S} = 1\mathbf{M}$

Elliptic Operation	Notation	Input	$\rightarrow$	Output	Cost
Differential Addition	dADD	XZ	$\rightarrow$	XZ	$4\mathbf{M} + 2\mathbf{S} = 6\mathbf{M}$
Doubling	dDBL	XZ	$\rightarrow$	XZ	$3\mathbf{M} + 2\mathbf{S} = 5\mathbf{M}$

2.3 Twisted Edwards curves

In [13] Edwards introduced a new normal form for elliptic curves which, among other advantages, benefit from fast elliptic operations. These curves have been generalized by Bernstein et al. [2]. A new coordinate system with a faster group law was introduced in [14], and their usage in ECM was considered in [4,3].

Definition 2 (Twisted Edwards curve). *Let* K *be a field and let* $a, d \in K$ *such that* $ad(a - d) \neq 0$. *A twisted Edwards curve, denoted* $E^E_{a,d}$, *is an elliptic curve whose affine points are all* $(x, y) \in K^2$ *such that*

$$ax^2 + y^2 = 1 + dx^2y^2. \tag{3}$$

In practice, the fastest formulas are obtained using a combination of three coordinates systems denoted projective, completed and extended. Input and output points are always represented in extended or projective coordinates, whereas completed coordinates are mainly used as an internal format. In the following, we shall use the best known formula from [14] for point doubling and point addition on twisted Edwards curves. Besides, an important feature of twisted Edwards curves is the existence of an efficient formula for point tripling [6].

In this article, we only consider twisted Edwards curves with $a = -1$. These curves allow for faster arithmetic and enjoy good torsion properties with regard to their use in ECM [1]. (See Section 2.5 for more details.) The input and output formats as well as the costs of the different elliptic operations that will be considered in the following are summarized in Table 2.

Table 2. Arithmetic cost of elliptic operations for twisted Edwards curves with $a = -1$ under the assumption $1\mathbf{S} = 1\mathbf{M}$

Elliptic Operation	Notation	Input	$\rightarrow$	Output	Cost
Addition	$\mathrm{ADD}_{\mathrm{comp}}$	ext.	$\rightarrow$	comp.	$4\mathbf{M}$
	ADD	ext.	$\rightarrow$	proj.	$7\mathbf{M}$
	$\mathrm{ADD}_{\varepsilon}$	ext.	$\rightarrow$	ext.	$8\mathbf{M}$
Doubling	DBL	ext. or proj.	$\rightarrow$	proj.	$3\mathbf{M} + 4\mathbf{S} = 7\mathbf{M}$
	$\mathrm{DBL}_{\varepsilon}$	ext. or proj.	$\rightarrow$	ext.	$4\mathbf{M} + 4\mathbf{S} = 8\mathbf{M}$
Tripling	TPL	ext. or proj.	$\rightarrow$	proj.	$9\mathbf{M} + 3\mathbf{S} = 12\mathbf{M}$
	$\mathrm{TPL}_{\varepsilon}$	ext. or proj.	$\rightarrow$	ext.	$11\mathbf{M} + 3\mathbf{S} = 14\mathbf{M}$

Contrary to Montgomery curves, twisted Edwards curves have a true elliptic addition. Hence the scalar multiplication can be computed using every generic algorithm available.

2.4 The best of both worlds

The best choice between Montgomery and Edwards curves for implementing the first stage of ECM depends on many parameters on top of which are the size of k (which depends on B_1), the memory available to store precomputed values, and the scalar multiplication algorithm used to compute $[k]P$. In this article, we exploit the best of both worlds by mixing twisted Edwards and Montgomery representations. We exploit a result from [2] which states that every twisted Edwards curve is birationally equivalent over its base field to a Montgomery curve. To the best of our knowledge, mixing Edwards and Montgomery representations was first suggested in [10] to speed-up arithmetic on elliptic curves in the x-coordinate-only setting. More recently, the idea has also been employed in the SIDH context [19].

Let K be a field with $\mathrm{char}(K) \neq 2$. According to [2, Theorem 3.2], every twisted Edwards curve $E^E_{a,d}$ defined over K is birationally equivalent over K to the Montgomery curve $E^M_{A,B}$, where $A = 2(a+d)(a-d)$ and $B = 4/(a-d)$. The map

$$(x, y) \mapsto ((1+y)/(1-y), (1+y)/(1-y)x) \tag{4}$$

is a birational equivalence from $E^E_{a,d}$ to $E^M_{A,B}$. (see [2, page 4] for a proof.) Using this map, we define a partial addition formula, denoted $\mathrm{ADD_M}$, which takes two points in extended coordinates on a twisted Edwards curve and computes their sum in XZ coordinates on the equivalent Montgomery curve. We express $\mathrm{ADD_M}$ as the composition of the group law on the completed twisted Edwards curve $\overline{E}^E_{a,d}$ and a partial conversion map, where

$$\overline{E}^E_{a,d} = \{((X:Z),(Y:T)) \in \mathbf{P}^1 \times \mathbf{P}^1 : aX^2T^2 + Y^2Z^2 = Z^2T^2 + dX^2Y^2\}.$$

(See [8] for more details on completed twisted Edwards curves and [3] for their usage in the ECM context.)

Given two points in extended coordinates on a twisted Edwards curve, the elliptic operation denoted $\mathrm{ADD_{comp}}$ computes their sum in completed coordinates $((X:Z),(Y:T))$ in $4\mathbf{M}$. Then, given that point in completed coordinates, one gets a representative in extended coordinates in another $4\mathbf{M}$ using the map

$$((X:Z),(Y:T)) \mapsto (XT:YZ:ZT:XY). \tag{5}$$

If only projective coordinates are needed, one simply omits the product XY. Observe that in the addition formulas on twisted Edwards curves introduced in [14] and recorded in Bernstein and Lange's Explicit Formula Database [7], the completed coordinates of the sum correspond to the four intermediate values $((E:G),(H:F))$.

For $X \neq 0$, the composition of the maps (4) and (5) is well defined. The map

$$((X:Z),(Y:T)) \longmapsto (T+Y : Z(T+Y)/X : T-Y) \tag{6}$$

sends points on a completed twisted Edwards curve with $X \neq 0$ to projective points on the equivalent Montgomery curve. For XZ coordinates, one simply omits the second coordinate:

$$((X:Z),(Y:T)) \longmapsto (T+Y :: T-Y). \tag{7}$$

Therefore, when defined, the operation $\mathrm{ADD_M}$, which takes as input two points of a twisted Edwards curves in extended coordinates and computes their sum on the equivalent Montgomery curves in XZ coordinates, costs only $4\mathbf{M}$ (see Table 3).

Let us now focus on the points for which (6) is not defined. The completed points with $X = 0$ correspond on $E^E_{a,d}$ to $(0,-1)$, the affine point of order 2, or $(0,1)$ the point at infinity. Using (7), the completed point $((0:1),(-1,1))$ of order 2 is mapped to the point $(0::1)$ of order 2 on the Montgomery curve. In this case, the mapping to XZ coordinates is thus well defined.

However, the map (7) sends the completed point at infinity $((0:1),(1:1))$ to the point $(2::0)$ on the Montgomery curve, which is different from $(0:1:0)$, the point at infinity on $E^M_{A,B}$. Nevertheless, in our context, using the point $(2::0)$ in place of the point at infinity $(0::0)$ is sufficient. Indeed, for all prime p dividing N, if the computations on the Edwards curve produce the point at infinity modulo p, what is important is that the map (7) returns a point on the equivalent Montgomery curve with $Z \equiv 0 \pmod{p}$. From the formulas for dDBL and dADD, it is easy to see that the remaining computations on the Montgomery curve also produce a point with $Z \equiv 0 \pmod{p}$. We have thus preserved the fact that p should divide the greatest common divisor between N and the Z-coordinate of the final point.

Table 3. New elliptic operation to switch from twisted Edwards curves to Montgomery curves

Elliptic Operation	Notation	Input $\rightarrow$ Output	Cost
Add & Switch	$\mathrm{ADD_M}$	Twisted Edwards ext. $\rightarrow$ Montgomery XZ	$4\mathbf{M}$

Note that after an $\mathrm{ADD_M}$, moving back to the twisted Edwards curve is not possible since the map (7) is not invertible, as the Y coordinate on the Montgomery curve is "lost". Nevertheless, as will be explained in Section 3, we easily cincumvert this obstacle by processing all the computations on the twisted Edwards curve before moving to the equivalent Montgomery curve for finalizing the scalar multiplication. Therefore, we never need to convert a point from XZ Montgomery back to the equivalent twisted Edwards curve.

2.5 Parameterization

In order to improve the probability of success of the ECM algorithm, we need to be able to generate curves with good torsion properties. Infinite families of curves with a rational or elliptic parametrization are used in the context of ECM to generate many different curves. Many of the best families for twisted Edwards curves have $a = -1$. So, restricting ourselves to twisted Edwards curves with $a = -1$ not only improves the arithmetic cost but also allows us to use curves with good torsion properties [1].

In practice, we use the parameterization from [3, Theorem 5.4] to generate a twisted Edwards curve with $a = -1$. We only need to compute the coefficients of the starting point in extended coordinates as the curve parameter d is never used in the formulæ. For the equivalent Montgomery curve, we solely compute the curve coefficient A since B is never used in the formulæ.

Using the parametrization from [3, Theorem 5.4] prevents us from choosing the curve coefficient A of the associated Montgomery curve. By choosing A such that $(A + 2)/4$ is small, we could have replaced, in the doubling formula, a

multiplication by a multiplication by a small constant. In our context, i.e., using medium-size integers, the arithmetic gain is not significant. Thus, we favored better torsion properties over a slightly lower theoretical arithmetic cost.

3 Scalar multiplication

After choosing a smoothness bound B_1 and a point P on an elliptic curve E, the core of the first stage of ECM consists of multiplying P by the scalar

$$k = \mathrm{lcm}(2, 3, 4, \ldots, B_1) = \prod_{p \text{ prime} \leq B_1} p^{\lfloor \log_p(B_1) \rfloor}$$

An elementary algorithm for computing $[k]P$ thus consists of performing, for each prime $p \leq B_1$, exactly $\lfloor \log_p(B_1) \rfloor$ scalar multiplications by p. These scalar multiplications may be computed using any addition chain compatible with the chosen curve E. If one uses the traditional binary addition chain, the number of point doublings depends on the bitlength of p, whereas the number of point additions is determined by its Hamming weight $w(p)$. Reducing the number of point additions by lowering the density of non-zero digits in the representation of the scalar is the core of many efficient scalar multiplication algorithms.

In the ECM context, the scalar k is entirely determined by the smoothness bound B_1. We may therefore derive much more efficient algorithms for computing $[k]P$. For example, instead of considering the primes p_i one at a time, one may multiply some of them together such that the weight of the product $w(\prod_i p_i)$ is lower than the sum of the individual weights $\sum_i w(p_i)$. This idea was first proposed by Dixon and Lenstra [12]. As an example, they give three primes $p_1 = 1028107$, $p_2 = 1030639$, $p_3 = 1097101$, of respective Hamming weights 10, 16 and 11, such that their product has Hamming weight 8. Beyond this example, the idea is advantageous only if one can find "good" recombinations for all the prime factors of k. Dixon and Lenstra used a greedy approach to find combination of primes by triples and managed to divide the overall number of point additions by roughly three. At that time, finding such a partition of the multiset $\mathcal{M}_{B_1}$ by triples was the best they could hope for. Surprisingly, they did not consider signed-digit representations to further reduce the overall cost. Clearly, their approach becomes unpractical for larger B_1 values and/or more general prime recombinations. Twenty years after Dixon and Lenstra's paper, Bos and Kleinjung managed to generalize the idea to arbitrary recombinations of primes and to extend its applicability to much larger B_1 values [9]. Considering all possible partitions of $\mathcal{M}_{B_1}$ being totally out of reach, they opted for the opposite strategy. A huge quantity of integers with very low density of non-zero digits in NAF form was first tested for smoothness. Then, among those integers that were B_1-powersmooth, a greedy algorithm was used to find a partition of $\mathcal{M}_{B_1}$ such that the cost of the resulting sequence of operations was minimal. For $B_1 = 256$, the best chain found led to a scalar multiplication algorithm which require 361

doublings and only 38 additions. The decomposition of $k = \mathrm{lcm}(2, \ldots, 256)$ into 15 batches of prime-products and their NAF expansions[2] are given in Table 4.

Table 4. An example of the best chain found for $B_1 = 256$ (see [9])

Batches of prime-products	NAF expansion	Cost
$23 \cdot 89$	$2^{11} - 2^0$	86 **M**
$83 \cdot 197$	$2^{14} - 2^5 - 2^0$	115 **M**
$191 \cdot 193$	$2^{15} + 2^{12} - 2^0$	122 **M**
$13 \cdot 19 \cdot 199$	$2^{15} + 2^{14} + 2^0$	122 **M**
$5 \cdot 13 \cdot 37 \cdot 109$	$2^{18} + 2^0$	135 **M**
$3^2 \cdot 7 \cdot 53 \cdot 157$	$2^{19} - 2^6 - 2^0$	150 **M**
$103 \cdot 137 \cdot 223$	$2^{21} + 2^{20} + 2^{10} + 2^0$	172 **M**
$5 \cdot 61 \cdot 149 \cdot 179$	$2^{23} - 2^{18} + 2^{13} - 2^0$	186 **M**
$3 \cdot 5 \cdot 29 \cdot 43 \cdot 113 \cdot 127$	$2^{28} - 2^0$	205 **M**
$3 \cdot 7 \cdot 11 \cdot 167 \cdot 173 \cdot 181$	$2^{30} + 2^{27} + 2^{11} + 2^0$	235 **M**
$3 \cdot 47 \cdot 59 \cdot 67 \cdot 73 \cdot 211$	$2^{33} - 2^{22} - 2^{19} + 2^8 + 2^6 - 2^0$	272 **M**
$11 \cdot 31 \cdot 79 \cdot 101 \cdot 131 \cdot 241$	$2^{36} + 2^{34} + 2^{18} + 2^2 + 2^0$	285 **M**
$17 \cdot 107 \cdot 139 \cdot 163 \cdot 229 \cdot 233$	$2^{41} - 2^{24} - 2^{13} - 2^9 - 2^0$	320 **M**
$41 \cdot 71 \cdot 97 \cdot 151 \cdot 227 \cdot 239 \cdot 251$	$2^{49} + 2^{44} + 2^{36} + 2^{32} - 2^3 - 2^0$	383 **M**
2^8	2^8	56 **M**
Total		2844 **M**

In the following, we shall use the term "block" to denote a batch of prime-products such as those given in Table 4. For each block, Dixon and Lenstra simply used addition chains, whereas Bos and Kleinjung took advantage of addition-subtraction chains through NAF decompositions. In this work, we consider more general decompositions in order to further reduce the overall cost. More precisely, we use three types of representations: double-base expansions, double-base chains (which contain NAF) and a subset of Lucas addition chains.

As an example, let us consider the primes $p_1 = 100003$, $p_2 = 100019$ and $p_3 = 109831$. Using the NAF decomposition, computing $[p_1]P$ requires 9 DBL, 7 DBL_ε, 6 ADD and 1 ADD_ε, resulting in 169 **M**. Similarly, $[p_2]P$ and $[p_3]P$ require 169 **M** and 168 **M** respectively. The NAF representation of their product only requires 447 **M**, i.e. 59 fewer multiplications than the cost of considering p_1, p_2 and p_3 independently.

[2] You may have observed that two of the given expansions do not satisfy the non-adjacent form, with two consecutive ones in their most significant positions. This is simply because evaluating $3P$ as $4P - P$ is more expensive than $2P + P$.

Let us now consider the following double-base representations of the same three primes. We have:

$$100003 = 2^{15}3^1 + 2^9 3^1 + 2^6 3^1 - 2^3 3^1 - 2^2 - 1 \tag{8}$$

$$100019 = 2^{15}3^1 + 2^9 3^1 + 2^6 3^1 - 2^2 3^1 - 1 \tag{9}$$

$$109831 = 2^{12}3^3 - 2^8 3^1 + 2^3 - 1 \tag{10}$$

Using (8), one may thus compute $[p_1]P$ with 10 DBL, 5 DBL_ε, 1 TPL, 4 ADD and 1 ADD_ε for a total cost of 158 **M**. Using (9) and (10), $[p_2]P$ and $[p_3]P$ requires 150 **M** and 145 **M** respectively. On twisted Edwards curves, the usage of triplings is thus already advantageous. Yet, the following double-base chain for their product

$$(((((2^2 3^1 + 1)2^6 3^1 - 1)2^{14}3^3 - 1)2^4 3^1 - 1)2^4 3^3 - 1)2^4 3^1 - 1, \tag{11}$$

leads to a chain for computing $[p_1 p_2 p_3]P$ with 28 DBL, 6 DBL_ε, 10 TPL, 6 ADD and 1 ADD_ε, for a total cost of 407 **M**. This represents an extra 40 **M** saving compared to the NAF-based approach.

In the next sections, we detail the generation of double-base expansions and double-base chains (which includes NAF) that are both compatible with twisted Edwards curves. We also present our strategy for generating a subset of Lucas chains for use with Montgomery curves.

3.1 Generation of double-base expansions

Let n be a positive integer, and let α, β be two pairwise integers. A double-base expansion of n can be seen as a partition of n into distinct parts of the form $\alpha^a \beta^b$. In this work, we solely consider the special case $(\alpha, \beta) = (2, 3)$ and we extend the usual notion of partition by allowing the parts to be either positive or negative, such that

$$n = \sum_{i=0}^{m} \pm 2^{d_i} 3^{t_i}, \tag{12}$$

where $(d_i, t_i) \neq (d_j, t_j)$ for every $0 \leq i < j \leq m$. Following the usual convention for integer partitions, we assume that the parts form a non-increasing[3] sequence so that $|2^{d_i}3^{t_i}| > |2^{d_j}3^{t_j}|$ for all $0 \leq i < j \leq m$. The length of a double-base expansion is equal to the number of parts in (12). Examples of double-base expansions of lengths 6, 5 and 4 respectively are given in (8), (9) and (10).

Given a double-base expansion for n as in (12), one can compute $[n]P$ with $D = \max_i d_i$ doublings, $T = \max_i t_i$ triplings and at most m additions using an algorithm by Meloni and Hasan [23]. Their algorithm is inspired by Yao's method [25] and requires the evaluation and storage of at most m elliptic curve points.

In order to limit the amount of additional storage in the resulting algorithms, we generated double-base expansions with at most 4 terms, i.e. for m varying

[3] in this case a decreasing sequence since the parts are distincts.

from 1 to 3. In practice, the memory requirements for the resulting algorithms are very low (see Section 5) and comparable to Bos and Kleinjung's low storage setting.

In fact, setting such a low value for the maximal length of double-base expansions was necessary to reduce computational workload. Indeed, without any restrictions, the total number of double-base chains with $m+1$ terms and such that $D \leq D_{\max}$ and $T \leq T_{\max}$ is equal to

$$2^{m+1}\binom{(D_{\max}+1)(T_{\max}+1)}{m+1}.$$

In our context, it was clearly more appropriate to let D and T cover larger ranges than to increase m. In Table 5, we give the parameters for m, D and T that we considered.

A few observations can be made to avoid generating the same double-base expansion more than once. First, notice that a double-base expansion for n immediately provides a double-base expansion for $-n$ by switching the sign of all the parts in (12). Hence, by imposing the sign of one of the terms, we generated only double-base expansions for positive integers; hence dividing the work effort by a factor two. We also noticed that a double-base expansion for n is easily converted into a double-base expansion for any integer of the form $n \times 2^a 3^b$, by adding a (resp. b) to each d_i (resp. t_i). Therefore, we only generated double-base expansions whose terms have no common factors. Given $D > 0$ and $T > 0$, the number of double-base expansions of length $m+1$ satisfying the above conditions can be computed exactly using a classical inclusion-exclusion principle. For completeness, we give the exact formula in Appendix A. In Table 5, we give the total number of double-base expansions that we generated for $m = 1, 2, 3$ and different intervals for D and T. For each double-base expansion, we tested the corresponding integer for 2^{13}-powersmoothness. We then evaluated the cost of Meloni and Hasan's scalar multiplication algorithm for those remaining double-base expansions. Unlike NAF decompositions, the double-base number system is highly redundant. For each value of m, we removed duplicates by keeping only double-base expansions of minimal cost. Yet, there might still exist duplicates for different values of m. Finally, we observed that it is always faster to process the powers of 2 after switching to Montgomery XZ coordinates. Thus, in order to reduce memory and speed-up the combination step (see Section 4), we filtered out all blocks corresponding to even integers. In Table 5, the column #db-exp gives the numbers of different double-base expansions that we generated for each value of m, the column #pow.smooth is the number of those expansions which corresponded to B_1-powersmooth integers, and the column #uniq (odd) accounts only for expansions of minimal costs corresponding to odd integers.

As seen in Table 5, for $m = 3$, we had to drastically reduce the upper bounds on D and T. Indeed, allowing D and T to span the intervals of values used for $m < 3$ would have required the generation of around $1.84 \cdot 10^{14}$ expansions. Nonetheless, in order to generate more integers of potential interest, we considered a subset of double-base expansions, namely double-base chains.

Table 5. Data on generated double-base expansions for $B_1 = 2^{13}$

m	D	T	#db-exp	#pow.smooth	#uniq (odd)	CPU time
1	0 – 255	0 – 127	$1.30 \cdot 10^5$	$1.15 \cdot 10^3$	$1.06 \cdot 10^3$	0 h
2	0 – 255	0 – 127	$6.37 \cdot 10^9$	$4.09 \cdot 10^5$	$2.97 \cdot 10^5$	3 h
3	0 – 128	0 – 64	$3.04 \cdot 10^{12}$	$1.64 \cdot 10^8$	$9.04 \cdot 10^7$	1048 h
Total			$3.04 \cdot 10^{12}$	$1.64 \cdot 10^8$		1051 h

3.2 Generation of double-base chains

A double-base chain for n is a double-base expansion as in (12) with divisibility conditions on the parts. More precisely, we impose that $2^{d_i}3^{t_i} \succeq 2^{d_{i+1}}3^{t_{i+1}}$ for $i \geq 0$, where $\succeq$ denotes the divisibility order, i.e. $x \succeq y \iff y|x$. All the double-base expansions given in the previous example are in fact double-base chains. The use of double-base chains for elliptic curve scalar multiplication was first introduced by Dimitrov et al. [11].

Given a double-base chain for n, one can compute $[n]P$ with m additions, $D = d_0$ doublings and $T = t_0$ triplings using a natural decomposition à la Horner as in (11). Unlike double-base expansions, the subsequent scalar multiplication algorithm does not require any additional storage.

The divisibility condition on the parts allows us to generate double-base chains for much larger values for m, D and T. As for double-base expansions, we only generated double-base chains for positive integers by fixing the sign of the first part $2^{d_0}3^{t_0}$. We also restricted our generation to double-base chains whose terms have no common factors, i.e. such that the smallest part $2^{d_m}3^{t_m} = \pm 1$. Under these conditions, the number of double-base chains with exactly D doublings, T triplings and m additions is given by:

$$2^m \sum_{i=0}^{m-1} (-1)^{m-i+1} \binom{m}{i+1} \binom{D+i}{D} \binom{T+i}{T}.$$

In Table 6 we give the number of double-base chains that we generated for different set of parameters m, D and T. Observe that double-base chains with $T = 0$ correspond to NAF expansions. In total, we generated more than $2.57 \cdot 10^{13}$ double-base chains, among which $2.29 \cdot 10^{10}$ corresponded to B_1-powersmooth integers, in approximately 9000 CPU hours.

3.3 Generation of Lucas chains

As seen in Section 2.2, Montgomery curves only admit a differential addition. Therefore the previous constructions (double-base expansions and chains) cannot be used to perform scalar multiplication. Instead, one uses Lucas chains.

Let n be a positive integer. A Lucas chain of length ℓ for n is a sequence of integers $(c_0, c_1, \ldots, c_\ell)$ such that $c_0 = 1$, $c_\ell = n$, and for every $1 \leq i \leq \ell$, either

Table 6. Data on generated double-base chains with smoothness bound 2^{13}

m	D	T	#db-chains	#pow.smooth	#uniq (odd)	CPU time
1	0 – 255	0 – 127	$6.55 \cdot 10^4$	$4.35 \cdot 10^2$	$3.93 \cdot 10^2$	0 h
2	0 – 255	0 – 127	$1.09 \cdot 10^9$	$3.82 \cdot 10^4$	$2.82 \cdot 10^4$	1 h
3	0 – 220	0 – 110	$3.41 \cdot 10^{12}$	$2.67 \cdot 10^6$	$1.54 \cdot 10^6$	1653 h
3	221 – 255	0	$7.84 \cdot 10^6$	0	0	0 h
4	0 – 75	0 – 40	$3.20 \cdot 10^{12}$	$1.43 \cdot 10^8$	$5.99 \cdot 10^7$	1013 h
4	76 – 255	0	$2.73 \cdot 10^9$	$5.46 \cdot 10^2$	$3.12 \cdot 10^2$	1 h
5	0 – 50	0 – 10	$2.86 \cdot 10^{11}$	$1.86 \cdot 10^9$	$4.25 \cdot 10^8$	68 h
5	51 – 255	0	$2.76 \cdot 10^{11}$	$2.98 \cdot 10^5$	$2.06 \cdot 10^5$	121 h
6	0 – 25	0 – 10	$2.35 \cdot 10^{11}$	$1.68 \cdot 10^{10}$	$9.04 \cdot 10^8$	171 h
6	26 – 200	0	$5.27 \cdot 10^{12}$	$3.01 \cdot 10^7$	$1.33 \cdot 10^7$	2204 h
7	0 – 115	0	$5.61 \cdot 10^{12}$	$3.68 \cdot 10^8$	$1.19 \cdot 10^8$	1596 h
8	0 – 80	0	$7.42 \cdot 10^{12}$	$3.66 \cdot 10^9$	$9.09 \cdot 10^8$	2240 h
Total			$2.57 \cdot 10^{13}$	$2.29 \cdot 10^{10}$		9068 h

it exists $j < i$ such that $c_i = 2c_j$ (doubling step), or there exist $j_0, j_1, j_d < i$ such that $c_i = c_{j_0} + c_{j_1}$ and $c_{j_d} = \pm(c_{j_0} - c_{j_1})$ (addition step).

Using a Lucas chain for n, $[n]P$ can be obtained by computing $[c_i]P$, for $1 \leq i \leq \ell$. When an addition step is encountered, the definition ensures that the difference of the two operands is already available. In general, Lucas chains are longer than binary, NAF, or double-base chains. Nevertheless, they sometimes lead to fast scalar multiplication algorithms since the cost of a differential addition is smaller than that of a plain addition.

The PRAC algorithm proposed by Montgomery [21] provides an efficient way to generate Lucas chains for any given integer n. It works by applying rules to a set of 3 points A, B and C, starting with $A = [2]P$, $B = C = P$. The rule to apply is chosen from a set of 9 rules based on two auxiliary integers d and e, starting with $d = n - \lfloor n/\phi \rceil$ and $e = 2\lfloor n/\phi \rceil - n$, where ϕ is the golden ratio. The two following invariants are maintained throughout the algorithm: $\pm C = A - B$ and $[n]P = [d]A + [e]B$.

We produced Lucas chains of length up to 13 by generating all possible combinations of PRAC rules up to that length. Observe that the nine rules from PRAC are not uniform regarding the type and number of curve operations they gather. For example, rule #2 consists of 2 doublings and 2 additions, whereas rule #4 only performs 1 addition. Consequently, the exhaustive generation of PRAC chains of length up to 13 allowed us to generate integers of size up to 26 bits. As there was lots of duplicates, we only kept the best Lucas chains for all odd integers before testing for smoothness. Data on the Lucas chains that we generated is given in Table 7.

Table 7. Data on generated Lucas chains for $B_1 = 2^{13}$. Only Lucas chains corresponding to odd integers were considered

#PRAC rules	#Lucas chains	#uniq	#pow.smooth	CPU time
13	$2.08 \cdot 10^{19}$	$1.25 \cdot 10^{7}$	$4.63 \cdot 10^{6}$	741 h

4 Combination of blocks for ECM stage 1

Let $\mathcal{B}$ be the set of all blocks generated with one of the method described in the previous section. For each block $b \in \mathcal{B}$, we define $n(b)$ as the integer associated to b, and $\mathcal{M}_b$ as the multiset composed of the prime factors (counted with multiplicity) of $n(b)$. We also define the arithmetic cost of b, denoted $\mathrm{cost}(b)$, as the sum of the costs of the elliptic operations used to compute the scalar multiplication by $n(b)$ using the algorithm associated to b. The arithmetic cost per bit, denoted $\mathrm{acpb}(b)$, is defined as $\mathrm{cost}(b)/\log_2(n(b))$.

By extension, we use the same notations for a set of blocks. Let $\mathcal{A} \subset \mathcal{B}$. Then $n(\mathcal{A}) = \prod_{b\in\mathcal{A}} n(b)$, $\mathcal{M}_\mathcal{A} = \bigcup_{b\in\mathcal{A}} \mathcal{M}_b$, $\mathrm{acpb}(\mathcal{A}) = \mathrm{cost}(\mathcal{A})/\log_2(n(\mathcal{A}))$. For the arithmetic cost, we need to take into account the switch from the twisted Edwards curve to the Montgomery curve, if necessary. Thus

$$\mathrm{cost}(\mathcal{A}) = \sum_{b\in\mathcal{A}} \mathrm{cost}(b) + \delta(\mathcal{A}) \underbrace{(\mathrm{cost}(\mathrm{ADD}_\mathrm{M}) - \mathrm{cost}(\mathrm{ADD}_\varepsilon))}_{-4\mathbf{M}},$$

where

$$\delta(\mathcal{A}) = \begin{cases} 1 \text{ if } \mathcal{A} \text{ contains at least 1 PRAC block} \\ 0 \text{ otherwise} \end{cases}$$

In practice, it is always cheaper to process the $\lfloor \log_2 B_1 \rfloor$ occurrences of the prime 2 in $\mathcal{M}_{B_1}$ using PRAC blocks. Therefore, we always switch from a twisted Edwards curve to the equivalent Montgomery curve at some point. Yet, the computations performed on the Montgomery curve are not restricted to the powers of 2. The PRAC blocks used in our best combinations often contains a few primes greater than 2 (see Table 8 and the data recorded at `http://eco.lirmm.net/double-base_ECM/`).

Let $B_1 > 0$ be the smoothness bound for ECM stage 1. The combination algorithms presented in the next sections consist of finding a subset $\mathcal{S}$ of $\mathcal{B}$ such that $\bigcup_{b\in\mathcal{S}} \mathcal{M}_b = \mathcal{M}_{B_1}$, or equivalently $\prod_{b\in\mathcal{S}} n(b) = k$, which minimizes $\mathrm{cost}(\mathcal{S})$.

4.1 Bos–Kleinjung algorithm

In 2012, Bos and Kleinjung describe a fast algorithm to compute a non-optimal solution (see [9, Algorithm 1]). The algorithm can be sketched as follow: start with $\mathcal{M} = \mathcal{M}_{B_1}$ and $\mathcal{S} = \emptyset$. Repeat until $\mathcal{M} \neq \emptyset$: pick the "best" block $b \in \mathcal{B}$ such that $\mathcal{M}_b \subseteq \mathcal{M}$ and the ratio $\mathrm{dbl}(b)/\mathrm{add}(b)$ is large enough (where $\mathrm{dbl}(b)$

and add(b) denote the number of doublings and additions in the NAF chain used to represent $n(b)$). Then, add b in $\mathcal{S}$ and subtract $\mathcal{M}_b$ from $\mathcal{M}$. Once the loop is exited, the algorithm returns $\mathcal{S}$. The bound on $\mathrm{dbl}(b)/\mathrm{add}(b)$ can be decreased during the algorithm if no block satisfies both conditions.

At each iteration, the "best" block is chosen with the help of a score function. This function is defined to favor blocks whose multisets share many large factors with the current multiset $\mathcal{M}$ of remaining factors. For a multiset $\mathcal{M}$ and a block b such that $\mathcal{M}_b \neq \emptyset$ and $\mathcal{M}_b \subseteq \mathcal{M}$, the score function is defined by[4]:

$$\mathrm{score}(b, \mathcal{M}) = \sum_{\substack{\ell=1 \\ a_\ell(\mathcal{M})\neq 0}}^{\lceil \log_2(\max(\mathcal{M}))\rceil} \frac{a_\ell(\mathcal{M}_b)}{a_\ell(\mathcal{M})}, \tag{13}$$

where

$$a_\ell(\mathcal{M}) = \frac{\#\{p \in \mathcal{M} \mid \lceil \log_2(p)\rceil = \ell\}}{\#\mathcal{M}}.$$

By default, the "best" block is the one which minimizes the score function.

In [9], a randomized version of the algorithm is also presented. The randomization is used to generate lots of different sets of solution and, hopefully, to improve the cost of the best one. Given an integer $0 < x < 1$, the randomized version selects the block with the smallest score with probability x or, with probability $1 - x$, skip it and repeat this procedure for the block with the second smallest score and so on.

4.2 Our algorithm

In a recent work, the authors of [15] replaced the ratio $\mathrm{dbl}(b)/\mathrm{add}(b)$ from Bos and Kleinjung's algorithm by the function

$$\kappa(b) = \frac{\log_2(n(b))}{\mathrm{dbl}(b) + 8/7\,\mathrm{add}(b) - \log_2(n(b))},$$

in order to take into account the bitlength of $n(b)$. This function κ produces slightly better results than Bos and Kleinjung's algorithm (see Table 9 in the "no storage" context). Yet, it is not readily adapted to our setting since it makes it difficult to take into account the costs of triplings and the fact that we use both twisted Edwards and Montgomery curves. For our combination algorithm, we consider a more generic function based on the arithmetic cost per bit of a block (acpb) as defined at the beginning of Section 4. Notice that on twisted Edwards curves, the function κ used by the authors of [15] is closely related to the arithmetic cost per bits of a NAF block. Indeed, we have $\mathrm{acpb}(b) \simeq (7\,\mathrm{dbl}(b) + 8\,\mathrm{add}(b))/\log_2(n(b)) = 7/\kappa(b) + 7$.

For our combination algorithm, we decided not to use the score function from Bos and Kleinjung's algorithm as we observed that it does not always

[4] There is a small mistake in the definition given in [9] which we were able to correct thanks to the examples following the definition.

achieve its goal to favor blocks with many large factors. For example, let us consider $B_1 = 256$ and two blocks b_1 and b_2 such that $\mathcal{M}_{b_1} = \{233, 193, 163\}$ and $\mathcal{M}_{b_2} = \{233, 193, 179, 109, 103, 73\}$. We would like the score function to favor the block b_2 since it contains more factors of sizes similar to the size of the elements of b_1. Yet, using (13), one gets $\text{score}(b_1, \mathcal{M}_{B_1}) = 3.043$ and $\text{score}(b_2, \mathcal{M}_{B_1}) = 4.214$, which means that the algorithm would select b_1 instead of b_2. Moreover, if b_3 is the best block that we could imagine with $\mathcal{M}_{b_3} = \mathcal{M}_{B_1}$, then its score would be worse than the two previous one, with $\text{score}(b_3, \mathcal{M}_{B_1}) = 8$.

We observed that using an algorithm similar the Bos and Kleinjung's algorithm where we always choose the block with the smallest arithmetic cost per bit did not yield better results than [9] or [15]. Thus, we tried a more exhaustive approch. A complete exhaustive search was totally out of reach, even for not-so-large values of B_1. However, the information provided by the previous results allowed us to envisage a somewhat exhaustive strategy.

Recall that our goal is to construct a subset $\mathcal{S}$ of $\mathcal{B}$ which minimizes $\text{cost}(\mathcal{S})$ and satisfies $\bigcup_{b \in \mathcal{S}} \mathcal{M}_b = \mathcal{M}_{B_1}$.

In order to reduce the enumeration depth, our first heuristic was to bound the number of blocks in the solution set $\mathcal{S}$. This constraint is rather natural as we want to favor blocks with many factors.

To further speed up our combination algorithm, we try to reduce the width of each step in the enumeration. First, notice that we can very easily obtain an upper bound on the arithmetic cost of the best solution set, for example by running the algorithms from Bos and Kleinjung [9] or from Ishii et al. [15] using our set of blocks $\mathcal{B}$. Then, we can use the following observation. Let C be an upper bound on the arithmetic cost of the best solution set and let $\mathcal{S}_0 \subseteq \mathcal{B}$ be a partial solution, i.e., such that $\bigcup_{b \in \mathcal{S}_0} \mathcal{M}_b \subsetneq \mathcal{M}_{B_1}$. Then a solution set $\mathcal{S}$ containing $\mathcal{S}_0$ satisfies $\text{cost}(\mathcal{S}) < C$, only if $\mathcal{S} \setminus \mathcal{S}_0$ contains at least one block whose arithmetic cost per bit is not greater than

$$\text{acpb}_{\max} = \frac{C - (\text{cost}(\mathcal{S}_0) + (1 - \delta(\mathcal{S}_0))(\text{cost}(\text{ADD}_{\text{M}}) - \text{cost}(\text{ADD}_{\varepsilon})))}{\log_2(n(\mathcal{M}_{B_1})) - \log_2(n(\mathcal{S}_0))}. \quad (14)$$

If we build our solution sets by adding blocks by increasing value of their arithmetic cost per bit, Equation (14) provides an upper bound for the arithmetic cost per bit of the next block that can be added to a partial solution set. A pseudo-code version of our combination algorithm is described in Algorithm 1 and an implementation in C is available at `http://eco.lirmm.net/double-base_ECM/`.

In Table 8, we give the best combination produced with Algorithm 1 for $B_1 = 256$. The resulting scalar multiplication algorithm requires 96 fewer multiplications than the best result from Bos and Kleinjung (see Table 4).

5 Results and comparison

In this section, we compare the cost of our implementation of ECM with the following implementations:

Algorithm 1 combine

Input: a set of blocks $\mathcal{B}$, a positive integer B_1, a bound ℓ on the length and an upper bound C on the arithmetic cost

Output: the solution set $\mathcal{S}$ with minimal cost such that $\#\mathcal{S} \leq \ell$, $\text{cost}(\mathcal{S}) < C$ and $\mathcal{M}_{\mathcal{S}} = \mathcal{M}_{B_1}$; or FAILURE if not such set exists

function ENUM_REC($\mathcal{S}_0$, $\mathcal{M}_{\text{rem}}$, $\mathcal{B}_{\text{rem}}$)
  $\mathcal{S} \leftarrow$ FAILURE ▷ by convention, we define cost(FAILURE) to be C
  $\mathcal{S} \leftarrow \mathcal{S}_{\text{new}}$
  $\mathcal{B}_{\text{new}} \leftarrow \mathcal{B}_{\text{rem}}$
  $\text{acpb}_{\max} \leftarrow$ value obtained using Equation (14) with $\mathcal{S}_0$, $\mathcal{M}_{B_1}$ and C
  for all $b \in \mathcal{B}_{\text{rem}}$ **do**
    if $\text{acpb}(b) > \text{acpb}_{\max}$ **then**
      break from the for loop
    else if $\mathcal{M}_b \subseteq \mathcal{M}_{\text{rem}}$ **then**
      $\mathcal{S}_{\text{new}} \leftarrow \mathcal{S}_0 \cup \{b\}$
      $\mathcal{M}_{\text{new}} \leftarrow \mathcal{M}_{\text{rem}} \setminus \mathcal{M}_b$
      if $\mathcal{M}_{\text{new}} = \emptyset$ and $\text{cost}(\mathcal{S}_{\text{new}}) < \text{cost}(\mathcal{S})$ **then**
        $\mathcal{S} \leftarrow \mathcal{S}_{\text{new}}$
      else if $\mathcal{M}_{\text{new}} \neq \emptyset$ and $\#\mathcal{S}_{\text{new}} < \ell$ **then**
        $\mathcal{S}_{\text{rec}} \leftarrow$ ENUM_REC($\mathcal{S}_{\text{new}}$, $\mathcal{M}_{\text{new}}$, $\mathcal{B}_{\text{new}}$)
        if $\text{cost}(\mathcal{S}_{\text{rec}}) < \text{cost}(\mathcal{S})$ **then**
          $\mathcal{S} \leftarrow \mathcal{S}_{\text{rec}}$
    $\mathcal{B}_{\text{new}} \leftarrow \mathcal{B}_{\text{new}} \setminus \{b\}$
  return $\mathcal{S}$

Sort $\mathcal{B}$ by increasing value of acpb
return ENUM_REC($\emptyset$, $\mathcal{M}_{B_1}$, $\mathcal{B}$)

Table 8. The best set of blocks computed with our algorithm for $B_1 = 256$. Type c corresponds to double-base chains, type e to double-base expansions and type m to blocks processed on the Mongomery model.

Blocks	Type		Cost
$193 \cdot 127 \cdot 109 \cdot 107 \cdot 61 \cdot 13 \cdot 7$	c	$2^{12} \cdot 3^{18} - 1$	309 **M**
$151 \cdot 31 \cdot 7$	c	$2^{15} - 1$	114 **M**
$227 \cdot 73 \cdot 67 \cdot 17$	c	$2^{21} \cdot 3^2 + 1$	180 **M**
$167 \cdot 149 \cdot 5$	c	$2^9 \cdot 3^5 - 1$	132 **M**
$251 \cdot 43 \cdot 41$	c	$2^{14} \cdot 3^3 + 2^4 \cdot 3^2 + 1$	151 **M**
$241 \cdot 229 \cdot 19$	c	$2^{20} + 2^4 - 1$	157 **M**
$211 \cdot 139 \cdot 13 \cdot 11$	c	$2^{22} - 2^8 - 1$	171 **M**
$233 \cdot 191 \cdot 173 \cdot 157$	c	$2^{27} \cdot 3^2 + 2^{18} \cdot 3 - 1$	230 **M**
$223 \cdot 137 \cdot 103 \cdot 83 \cdot 37$	c	$2^{30} \cdot 3^2 + 2^{11} - 1$	251 **M**
$179 \cdot 101 \cdot 97 \cdot 47 \cdot 29 \cdot 23 \cdot 5$	c	$2^{38} - 2^3 - 1$	283 **M**
$181 \cdot 131 \cdot 89 \cdot 59 \cdot 11$	c	$2^{24} \cdot 3^4 + 2^{17} \cdot 3^4 - 2^8 - 1$	241 **M**
$239 \cdot 199 \cdot 197 \cdot 163 \cdot 113 \cdot 79 \cdot 71 \cdot 53$	e	$2^{46} \cdot 3^6 + 2^{42} + 2^{14} + 3^3$	421 **M**
Switch to Montgomery. Last addition in the above block is an $\mathrm{ADD_M}$			-4 **M**
$5 \cdot 3^5$	m		72 **M**
2^8	m		40 **M**
Total			2748 **M**

- the ECM code inside CADO-NFS [24] (version 2.3),
- the software EECM-MPFQ [5],
- the article "ECM at Work" [9],
- the article [15], referenced as "ECM on Kalray" in the following.

The ECM code inside CADO-NFS is the only implementation that uses Montgomery curves. The other three use twisted Edwards curves with $a = -1$. For "ECM at work", we consider the two settings called no storage and low storage as presented in the article. The cost comparison for the stage 1 of ECM is given in Table 9. In Figure 1, we compare the arithmetic cost per bit of these various implementations for more values of B_1.

For curves with the same torsion as the ones we use (see Section 2.5), the stage 1 of GMP-ECM is implemented with Montgomery curves and uses the same algorithms as CADO-NFS. Thus, for theses curves, the stage 1 of GMP-ECM and CADO-NFS have the same cost.

Regarding storage requirements, our implementation is also competitive. For the blocks using Lucas chains on Montgomery curves, we only need three extra points in XZ coordinates, in addition to the input and output points. For double-base chains, we do not need any extra point and for double-base expansions, as we only generated expansions with at most 4 terms, we need at most 4 additional

Table 9. Number of modular multiplication (**M**) for various implementations of ECM (stage 1) and some commonly used smoothness bounds B_1 assuming $1\mathbf{S} = 1\mathbf{M}$

$B_1 =$	256	512	1024	8192
CADO-NFS [24]	3091	6410	12916	104428
EECM-MPFQ [5]	3074	6135	12036	93040
ECM at Work (no storage) [9]	2844	5806	11508	91074
ECM on Kalray [15]	2843	5786	11468	90730
ECM at Work (low storage) [9]	2831	5740	11375	89991
this work	2748	5667	11257	89572

points in extended coordinates. So our storage requirements are similar to the low storage setting of [9] and much lower than the hundred of points required by EECM-MPFQ.

We note that the fact that the output of stage 1 is a point in XZ coordinates on a Montgomery curve is not a burden for the stage 2 of the ECM algorithm. The stage 2 from [15,9] is computed using a baby-step giant-step algorithm. A complete description in the case of twisted Edwards curves is given in [20, Section 3.2]. In CADO-NFS, the stage 2 also uses a baby-step giant-step algorithm, but on Montgomery curves. Using the same approach, we managed to greatly reduce the cost of stage 2 thanks to using finer parameters adjustments. More precisely, the baby-step giant-step method for stage 2 of ECM is parameterized by a value called ω, which in CADO-NFS was set to a constant value. We observed that adjusting ω according to the values of B_1 and B_2 yields significant speed-ups for large values of B_1 and B_2. The costs of the stage 2 for these different implementations are given in Table 10.

Table 10. Number of modular multiplications (**M**) for ECM stage 2 assuming $1\mathbf{S} = 1\mathbf{M}$

$B_1 =$	256	512	1024	8192
$B_2 =$	2^{14}	$3 \cdot 2^{14}$	$7 \cdot 2^{14}$	$80 \cdot 2^{14}$
CADO-NFS [24]	2387	6120	13264	134761
ECM on Kalray [15] (based on [20])	2538	5812	11410	91122
this work	2227	5160	10273	89866

In order to evaluate the practical impact of our approach, we implemented our new algorithms for scalar multiplications in CADO-NFS. To assess the efficiency for large composite numbers, we run a small part of the sieving phase for RSA-200 and RSA-220 with the default parameters and observed that the

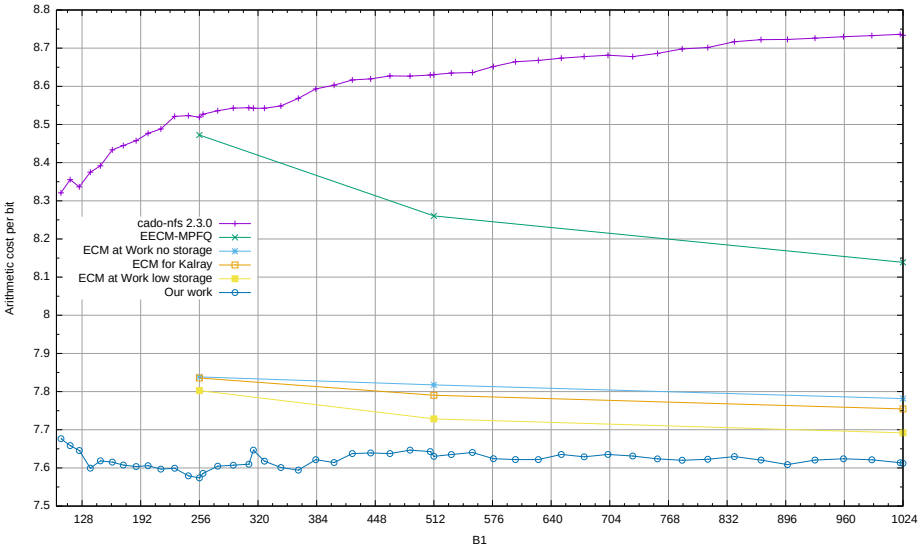

Fig. 1. Arithmetic cost per bit for the scalar multiplication of ECM stage 1 of ECM assuming $1\mathbf{S} = 1\mathbf{M}$.

cofactorization time decreased by 5 % to 10 %, in accordance with our theoretical estimates.

6 Conclusion

In the context of NFS cofactorization, ECM is used to break into primes billions of medium-size integers. In practice, only a few B_1-values are used, making it possible to precompute almost optimal algorithms for these customary B_1-values. Following the works from Dixon and Lenstra and Bos and Kleinjung, we generated over 10^{19} chains of various types and combined them using a quasi exhaustive approach. Our implementation uses both twisted Edwards curves, through efficient double-base decompositions, and Montgomery curves. For switching from one model to the other, we introduced a partial addition-and-switch operation which computes the sum in Montgomery XZ coordinates of two points given on an equivalent Edwards curve.

For $B_1 \leq 8192$, our implementation requires fewer modular multiplications than any other publicly available implementation of ECM. It requires significantly less memory than EECM-MPFQ. The arithmetic cost per bit of our implementation is relatively stable, around $7.6\,\mathbf{M}$. Extending the current approach based on prime batches and recombination, for example by considering extended double-base expansions and chains, is possible. Yet, significant speed-ups seems difficult to prefigure. For larger B_1-values, our combination algorithm is likely

to become unpractical. However, it can be used iteratively to extend any current best combination results.

References

1. Barbulescu, R., Bos, J.W., Bouvier, C., Kleinjung, T., Montgomery, P.: Finding ECM-friendly curves through a study of Galois properties. In: ANTS X: Proceedings of the Tenth Algorithmic Number Theory Symposium. Open Book Series, vol. 1, pp. 63–86 (2013). https://doi.org/10.2140/obs.2013.1.63
2. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards curves. In: Progress in Cryptology – AFRICACRYPT 2008. Lecture Notes in Computer Science, vol. 5023, pp. 389–405. Springer (2008)
3. Bernstein, D.J., Birkner, P., Lange, T.: Starfish on strike. In: Progress in Cryptology – LATINCRYPT 2010. Lecture Notes in Computer Science, vol. 6212, pp. 61–80. Springer (2010)
4. Bernstein, D.J., Birkner, P., Lange, T., Peters, C.: ECM using Edwards curves. Mathematics of Computation **82**, 1139–1179 (2013)
5. Bernstein, D.J., Birkner, P., Lange, T., Peters, C., et al.: EECM-MPFQ: ECM using Edwards curves, `http://eecm.cr.yp.to/index.html`
6. Bernstein, D.J., Chuengsatiansup, C., Lange, T.: Double-base scalar multiplication revisited. Cryptology ePrint Archive, Report 2017/037 (2017), `https://eprint.iacr.org/2017/037`
7. Bernstein, D.J., Lange, T.: Explicit-formulas database. `http://www.hyperelliptic.org/EFD/`, joint work by Daniel J. Bernstein and Tanja Lange, building on work by many authors.
8. Bernstein, D.J., Lange, T.: A complete set of addition laws for incomplete Edwards curves. Cryptology ePrint Archive, Report 2009/580 (2009), `https://eprint.iacr.org/2009/580`
9. Bos, J.W., Kleinjung, T.: ECM at work. In: Advances in Cryptology – ASIACRYPT 2012. pp. 467–484. No. 7658 in Lecture Notes in Computer Science, Springer (2012)
10. Castryck, W., Galbraith, S., Farashahi, R.R.: Efficient arithmetic on elliptic curves using a mixed Edwards–Montgomery representation. Cryptology ePrint Archive, Report 2008/218 (2008), `https://eprint.iacr.org/2008/218`
11. Dimitrov, V., Imbert, L., Mishra, P.K.: Efficient and secure elliptic curve point multiplication using double-base chains. In: Advances in Cryptology, ASIACRYPT 2005. Lecture Notes in Computer Science, vol. 3788, pp. 59–78. Springer (2005). https://doi.org/10.1007/11593447_4
12. Dixon, B., Lenstra, A.K.: Massively parallel elliptic curve factoring. In: Advances in Cryptology – EUROCRYPT' 92. Lecture Notes in Computer Science, vol. 658, pp. 183–193. Springer (1992)
13. Edwards, H.M.: A normal form for elliptic curves. Bulletin of the American Mathematical Society **44**, 393–422 (July 2007)
14. Hisil, H., Wong, K.K.H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 326–343. Springer (2008)
15. Ishii, M., Detrey, J., Gaudry, P., Inomata, A., Fujikawa, K.: Fast Modular Arithmetic on the Kalray MPPA-256 Processor for an Energy-Efficient Implementation of ECM. IEEE Transactions on Computers **66**(12), 2019–2030

(Dec 2017). https://doi.org/10.1109/TC.2017.2704082, https://hal.inria.fr/hal-01299697
16. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J.W., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., te Riele, H., Timofeev, A., Zimmermann, P.: Factorization of a 768-bit rsa modulus. In: Rabin, T. (ed.) Advances in Cryptology – CRYPTO 2010. Lecture Notes in Computer Science, vol. 6223, pp. 333–350. Springer (2010)
17. Lenstra, A.K., Lenstra, H.W. (eds.): The development of the number field sieve, Lecture Notes in Mathematics, vol. 1554. Springer (1993)
18. Lenstra, H.W.: Factoring integers with elliptic curves. Annals of Mathematics **126**(3), 679–673 (1987)
19. Meyer, M., Reith, S., Campos, F.: On hybrid SIDH schemes using Edwards and Montgomery curve arithmetic. Cryptology ePrint Archive, Report 2017/1213 (2017), https://eprint.iacr.org/2017/1213
20. Miele, A.: On the Analysis of Public-Key Cryptologic Algorithms. Ph.D. thesis, EPFL (2015)
21. Montgomery, P.L.: Evaluating recurrences of form $X_{m+n} = f(X_m, X_n, X_{m-n})$ via Lucas chains. (Dec 1983), unpublished
22. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation **48**(177), 243–264 (Jan 1987)
23. Méloni, N., Hasan, M.A.: Elliptic curve scalar multiplication combining Yao's algorithm and double bases. In: Cryptographic Hardware and Embedded Systems, CHES 2009. Lecture Notes in Computer Science, vol. 5747, pp. 304–316. Springer (2009)
24. The CADO-NFS Development Team: CADO-NFS, An Implementation of the Number Field Sieve Algorithm (2017), http://cado-nfs.gforge.inria.fr/, release 2.3.0
25. Yao, A.C.C.: On the evaluation of powers. SIAM Journal of Computing **5**(1), 100–103 (1976)
26. Zimmermann, P.: 50 largest factors found by ECM. https://members.loria.fr/PZimmermann/records/top50.html
27. Zimmermann, P., Dodson, B.: 20 years of ECM. In: Algorithmic Number Theory. ANTS 2006. Lecture Notes in Computer Science, vol. 4076, pp. 525–542. Springer (2006)
28. Zimmermann, P., et al.: GMP-ECM (elliptic curve method for integer factorization), https://gforge.inria.fr/projects/ecm/

A Counting double-base expansions

Number of double-base expansions of the form $n = 2^{d_0}3^{t_0} + \sum_{i=1}^{m} \pm 2^{d_i}3^{t_i}$ with $\max_i d_i = D$, $\max_i t_i = T$ and whose terms have no common factors:

$$2^m \left[\binom{(D+1)(T+1)}{m+1} - 2\binom{(D+1)T}{m+1} - 2\binom{(T+1)D}{m+1} + 4\binom{DT}{m+1} + \binom{(D+1)(T-1)}{m+1} + \binom{(T+1)(D-1)}{m+1} - 2\binom{(D-1)T}{m+1} - 2\binom{(T-1)D}{m+1} + \binom{(D-1)(T-1)}{m+1} \right]$$

The proof is omitted. It follows a classical inclusion-exclusion principle.