ECM at Work

Joppe W. Bos and Thorsten Kleinjung

Laboratory for Cryptologic Algorithms, EPFL, Lausanne, Switzerland

Abstract. The performance of the elliptic curve method (ECM) for integer factorization plays an important role in the security assessment of RSA-based protocols as a cofactorization tool inside the number field sieve. The efficient arithmetic for Edwards curves found an application by speeding up ECM. We propose techniques based on generating and combining addition chains to optimize Edwards ECM in terms of both performance and memory requirements. This makes our approach very suitable for memory-constrained devices such as graphics processing units. For commonly used ECM parameters we are able to lower the required memory up to a factor 55 compared to the state-of-the-art Edwards ECM approach.

Keywords: Elliptic curve factorization, cofactorization, addition chains, twisted Edwards curves, parallel architectures.

1 Introduction

Today, more than 25 years after its invention by Hendrik Lenstra Jr., the elliptic curve method [21] (ECM) remains the asymptotically fastest integer factorization method for finding relatively small prime factors of large integers. Although it is not the fastest general purpose integer factorization method, when factoring a composite integer $n = pq$ with $p \approx q \approx \sqrt{n}$ the number field sieve [28, 20] (NFS) is asymptotically faster, it has recently received a renewed research interest due to the discovery of an interesting normal form for elliptic curves introduced by Edwards [11].

From a cryptologic point of view the practical performance of ECM is important since it is used to rapidly factor many small (up to one or two hundred bits) integers inside NFS. This is illustrated by the fact that it is estimated that five to twenty percent of the total wall-clock time was spent in ECM in the current world-record factorization of a 768-bit RSA number [17] (and it is expected that this percentage will grow for larger factorizations). Using ECM as a tool to factor many small numbers inside NFS is an active research area by itself. Offloading this work to reconfigurable hardware such as field-programmable gate arrays is studied in [33, 26, 13, 9, 14, 22, 36] while [4, 3] considers parallel architectures such as graphics processing units and the Cell broadband engine architecture. A comparison between software and hardware based solutions is presented in [18].

Traditionally, ECM is implemented using Montgomery curves [23] and uses the various techniques described in [35]. The most-widely used ECM implementation is GMP-ECM [37] and this implementation, or modifications to it, is responsible for setting all recent ECM record factorizations [6]. After the invention of Edwards curves Bernstein et al. explored the possibility to use these curves in the ECM setting [2]. Hisil et al. [16] published a coordinate system for Edwards curves which results in the fastest known realization of curve arithmetic. A follow-up paper by Bernstein et al. discusses the usage of these "$a = -1$" twisted Edwards curves [1] for ECM. This speedup comes at a price, when using Edwards curves addition chains [31] equipped with large windowing sizes [7] are used (cf. [5] for a summary of these techniques). The memory requirement for Edwards ECM grows roughly linearly with the input parameters of ECM while a small constant number of residues modulo n are sufficient when using Montgomery curves.

In this paper we optimize ECM by exploiting the fact that the same scalar is often used when computing the elliptic curve scalar multiplication (ECSM), allowing one to prepare particularly good

addition chains for these fixed scalars. Our approach is inspired by the ideas used in the ECM implementation by Dixon and Lenstra [10] from 1992. In [10] the total cost to compute the ECSM, in terms of point duplications and point additions, is lowered by testing if the computation of the ECSM using batches of small prime products is cheaper (requires fewer point additions) than processing the primes one at a time (or all in one big batch). We generalize this idea: many billions of integers, which are constructed such that they can be computed using an addition chain with a high duplication/addition ratio, are tested for smoothness and factored. By fixing different popular elliptic curve scalar values used in ECM inside NFS we are able to combine some of these integers using a greedy approach. This results in a *more efficient* ECSM algorithm with a *smaller* memory footprint. To illustrate, compared to the cofactorization setting considered by Bernstein et al. in [4, 3] (using the parameter $B_1 = 2^{13}$) the techniques from this paper reduce the memory by a factor 55. This makes our approach particularly interesting for environments where the memory (per thread) is constrained; e.g. graphics processing units.

This paper is organized as follows. After recalling the preliminaries in Section 2 the notation and basic idea behind elliptic curve constant scalar multiplication is discussed in Section 3. Section 4 explains how to combine these chains such that they might result in a faster and more memory efficient ECM. Section 5 explains a side-effect why certain chains require more modular multiplications and Section 6 presents the obtained results. Section 7 concludes the paper.

2 Preliminaries

2.1 The Elliptic Curve Method

The elliptic curve method (ECM) for integer factorization [21] is analogous to the Pollard $p-1$ integer factorization method [29] and attempts to factor a composite integer $n = pq$ ($1 < p < q < n$). The general idea behind ECM is as follows (we follow the description from [21]). First, pick a random point P and construct an elliptic curve E over $\mathbf{Z}/n\mathbf{Z}$ such that $P \in E(\mathbf{Z}/n\mathbf{Z})$ (cf. [19, Sec. 2.B]). Next, compute the elliptic curve scalar multiplication $Q = kP \in E(\mathbf{Z}/n\mathbf{Z})$. The positive integer k is selected such that it is divisible by many small prime powers: e.g. $k = \mathrm{lcm}(1,2,\ldots,B_1)$ for some bound $B_1 \in \mathbf{Z}$. If the order $\#E(\mathbf{F}_p)$ is B_1-powersmooth (an integer is defined to be B-powersmooth if none of the prime powers dividing this integer is greater than B) then $\#E(\mathbf{F}_p) \mid k$. In other words, $Q = kP$ and the neutral element of the curve become the same modulo p. In this event, a failure occurred in the group operation defined for $E(\mathbf{Z}/n\mathbf{Z})$ and we have $p \mid \gcd(n, Q_z)$, where Q_z is the z-coordinate of the point Q when using projective coordinates. If $q \nmid \gcd(n, Q_z)$ then we have factored n.

Hasse proved (see e.g. [32, Theorem 1.1]) that the order $\#E(\mathbf{F}_p)$ is in the interval $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$. The advantage of ECM is that one can randomize by trying different curves, thus obtaining different orders. It has been shown in [21] that the (heuristic) run-time of ECM depends mainly on p, the smallest non-trivial prime divisor of n, and can be expressed as

$$O(\exp((\sqrt{2}+o(1))(\sqrt{\log p \log\log p}))M(\log n))$$

where $M(\log n)$ represents the complexity of multiplication modulo n and the $o(1)$ is for $p \to \infty$.

The approach described here is often referred to as "stage 1". There is a "stage 2" continuation for ECM which takes as input a bound $B_2 \in \mathbf{Z}$ and succeeds (in factoring n) if $Q = kP$ has prime order ℓ (for $B_1 < \ell < B_2$) in $E(\mathbf{F}_p)$. This means that $\#E(\mathbf{F}_p)$ is B_1-powersmooth except for one prime factor which is below B_2. There are several techniques [8, 23, 24] how to perform stage 2 efficiently. In the following we will focus on stage 1 only.

Table 1. Performance comparison between GMP-ECM and EECM-MPFQ in terms of modular multiplications (**M**) and squarings (**S**) together with the required number of residues modulo n (R) which needs to be kept in memory.

	GMP-ECM [37]				EECM-MPFQ [2]			
B1	**#S**	**#M**	**#S+#M**	*#R*	**#S**	**#M**	**#S+#M**	*#R*
256	1 066	2 025	3 091	14	1 436	1 638	3 074	38
512	2 200	4 210	6 410	14	2 952	3 183	6 135	62
1024	4 422	8 494	12 916	14	5 892	6 144	12 036	134
8192	35 508	68 920	104 428	14	47 156	45 884	93 040	550

2.2 Cofactorization using ECM

The relation collection phase, one of the two main phases of NFS, generates a lot of composite integers which need to be tested for powersmoothness. This is done using different factorization techniques and is denoted as the cofactorization phase. To illustrate, the total time spent in the cofactorization procedure was roughly one third of the sieving time when factoring the 768-bit RSA modulus in [17]. Note that this one third includes the time of pseudo primality tests and different factorization methods: quadratic sieve [30], Pollard $p-1$ [29] and ECM. In this cofactorization phase only composites up to 140 bits were considered and ECM was used only for composites up to 109 bits. The parameters for ECM varied depending on the size of the composites and ranged from $B_1 = 150$ to $B_1 = 500$ where often only a single curve was tried with a maximum of around eight curves. Observing the trend of past record factorizations, it is conceivable that cofactorization becomes more important in bigger factorizations (cf. [4] for more detailed arguments about the significance of ECM in NFS).

2.3 Montgomery versus Edwards Curves

The main motivation to use Edwards (over Montgomery) curves is performance. There are two implementations of ECM using Edwards curves available: GMP-EECM [2] using the "$a = 1$" Edwards curves and EECM-MPFQ [1] using the "$a = -1$" Edwards curves. The $a = -1$ Edwards ECM approach is the fastest in practice and we use this as the base setting to compare to. Table 1 compares the required number of multiplications and squarings required in GMP-ECM and EECM-MPFQ for different typical B_1 values used in ECM when used as a cofactorization method in NFS. These numbers show that using Edwards curves results in fewer modular multiplications and squarings. However, the required storage for GMP-ECM (Montgomery curves) is independent of B_1 while it grows almost linearly with the size of B_1 and is significantly higher, due to the use of width-w windowing methods, for EECM-MPFQ (Edwards curves, see [2, Table 4.1]).

3 Elliptic Curve Constant Scalar Multiplication

Most of the addition/subtraction chains based algorithms in practice use a w-bit windowing technique, for some (optimal) width w, to reduce the number of required elliptic curve additions. The total number of additions may be significantly reduced by using this approach but one also needs to store more points: 2^{w-1} when using sliding windows [34]. In environments where the available memory per thread is low, these methods cannot be used or one is forced to settle for a suboptimal window size. A prime example of such a platform are graphics processing units (GPUs); one of the latest GPU architectures [25] (Fermi) shares 64 kilobyte fast shared memory per 32 processors and each processor typically time-shares multiple threads (e.g., 16 to 32 corresponding to 128 to 64 bytes per thread).

We investigate two approaches to lower the number of elliptic curve additions *and* the storage required to compute the scalar product. Our approach is inspired by the results reported by Dixon and

Lenstra [10]. Suppose we have a scalar $k = \mathrm{lcm}(1,\ldots,B_1) = \prod_{i=1}^{\ell} p_i$, where the p_i are primes which can occur multiple times. Typically, the ECSM is implemented processing one such p_i at a time [35]. In [10] it is suggested to process the p_i in *batches*; i.e. multiply a batch of p_i's at a time such that the weight of the product $w(\prod_i p_i)$, the number of ones in the binary representation of $\prod_i p_i$, is (much) lower than the sum of the individual weights $\sum_i w(p_i)$. If this is the case then the number of required EC-additions is reduced when using the straight forward double-and-add approach (which does not require to store any additional precomputed points). Such low-weight products can be constructed by greedily searching through b-tuples of the p_i where b is small. In [10] b was at most 3 which reduced the total weight by approximately a factor three. As an example the following triple is given

$$1028107 \cdot 1030639 \cdot 1097101 = 1162496086223388673$$
$$w(1028107) = 10, \quad w(1030639) = 16, \quad w(1097101) = 11,$$
$$w(1162496086223388673) = 8,$$

where the product of primes of weights 10, 16, and 11 results in a integer of weight eight. The resulting composite integer can be computed using an addition chain requiring only seven additions and 60 duplications using the naive double-and-add algorithm.

In this section we explore different methods to find numbers which can be constructed using even better (higher) duplication/addition ratios. These methods do not aim to construct sequences by combining the different p_i (as in [10]) but we propose an opposite approach by factoring many integers which are the result of addition chains with high duplication/addition ratios and subsequently combining these integers such that all p_i's are used. These addition chains are constructed such that they do not require any large lookup tables. Notice that the information encoding the sequence of arithmetic operations has to be stored (in all approaches). This does not pose a problem since this information is constant and can be shared among all the computational units (or streamed to the units or even hardcoded) and hence does not result in additional overhead in practice.

In the remainder of the paper we denote addition chains using both additions and subtractions simply as addition chains.

3.1 Addition Chains With Restrictions

In order to generate integers which can be computed using an addition chain with a high duplication/addition ratio we need to construct and denote addition chains of a certain length m. An addition chain is a sequence of duplications, additions and subtractions denoted by D, A and S respectively. A duplication can always be assumed to apply to the previously generated element in the addition chain (instead of duplicating any previous element), since one can reorder the symbols such that duplication always occurs on the last element without changing the result of the addition chain. In some cases this might results in a shorter (more efficient) sequence when the same element is duplicated multiple times.

Let us define the set of symbols $\mathscr{O}$ as

$$\mathscr{O} = \{D\} \cup \{A_{i,j} \mid i,j \in \mathbf{Z}, i > j\} \cup \{S_{i,j} \mid i,j \in \mathbf{Z}, i > j\},$$

where the subscripts indicate on which element in the chain we compute (this is made more precise later). The set of all m-tuples, ordered lists of m elements, of symbols in $\mathscr{O}$ with the restriction that no elements can be used which have not yet been generated is

$$O_m = \{(o_{m-1},\ldots,o_0) \in \mathscr{O}^m \mid o_k \in \{D\} \cup \{A_{i,j} \mid i \leq k\} \cup \{S_{i,j} \mid i \leq k\}, 0 \leq k < m\}.$$

In order to construct an addition chain from such an m-tuple of symbols we define functions $\sigma_m : \mathcal{O} \times \mathbf{Z}^{m+1} \to \mathbf{Z}^{m+2}$ such that $(o,(t_m,\ldots,t_0)) \mapsto (t_{m+1},t_m,\ldots,t_0)$ where

$$t_{m+1} = \begin{cases} 2t_m & \text{if } o = D, \\ t_i + t_j & \text{if } o = A_{i,j}, \\ t_i - t_j & \text{if } o = S_{i,j}. \end{cases}$$

Given an m-tuple of symbols $(o_{m-1},\ldots,o_0) \in O_m$ the $(m+1)$-tuple of integers associated to this addition chain is $\sigma_{m-1}(o_{m-1},\sigma_{m-2}(o_{m-2},\ldots,\sigma_0(o_0,1)\ldots))$ and the *resulting integer* produced by this chain is t_m.

As an example consider the 7-tuple of symbols $(S_{6,0},D,D,A_{3,0},D,D,D) \in O_7$ which corresponds to the 8-tuple of integers in the addition chain $(35,36,18,9,8,4,2,1)$ computed as $\sigma_6(S_{6,0},\sigma_5(D,\sigma_4(D,\sigma_3(A_{3,0},\sigma_2(D,\sigma_1(D,\sigma_0(D,1)))))))$. The function σ_m is the correspondence between a tuple of symbols and the actual addition chain. The example shows how to compute the resulting integer 35 using one subtraction, one addition and five duplications.

The set of tuples O_m consists of the most generic type of addition chains, a significant amount of tuples corresponds to chains which perform useless (unnecessary) computations. An example is computing the addition (or subtraction) of two previous values without using this result. To address this we define a more restricted set of tuples $\mathcal{P}_m \subset O_m$ as

$$\mathcal{P}_m = \{(o_{m-1},\ldots,o_0) \in O_m \mid o_k \in \{D\} \cup \{A_{i,j} \mid i = k\} \cup \{S_{i,j} \mid i = k\}, 0 \le k < m\}.$$

These additional restrictions ensure that, just as for the duplication, we only add or subtract to the last integer in the sequence to obtain the next one. Such chains are known as *Brauer chains* or *star addition chains* [15, Section C6].

In this setting we write A_j and S_j for $A_{i,j}$ and $S_{i,j}$, respectively, and $k > 0$ subsequent instances of D are denoted by D^k. The previous example can now be written as $S_0D^2A_0D^3 \in \mathcal{P}_7$ by abusing the notation: omitting the brackets and comma's. In practice we would generate sequences of symbols such that a number of elliptic curve additions $\mathbf{A}$ and duplications $\mathbf{D}$ are fixed and look at sequences of symbols of length $m = \mathbf{A} + \mathbf{D}$ which use $\mathbf{A}$ times A_j or S_j and $\mathbf{D}$ times D. Different tuples might compute the same integer result. Using our example, the number 35 can be obtained with $\mathbf{D} = 5$ and $\mathbf{A} = 2$ in different ways

$$\begin{aligned} 35 &= (2^3+1)\cdot 2^2 - 1 \quad S_0D^2A_0D^3 \in \mathcal{P}_7 \\ &= (2^4+1)\cdot 2 + 1 \quad A_0DA_0D^4 \in \mathcal{P}_7. \end{aligned}$$

3.2 Generating Addition Chains

We discuss how to efficiently generate the resulting integers t_m in a low-storage and no-storage setting.

The Low-Storage Setting. Let $\mathbf{A}$ be the number of elliptic curve additions and $\mathbf{D}$ the number of elliptic curve duplications (with $\mathbf{D} \ge \mathbf{A}$). The generation of *all* the tuples in $\mathcal{P}_m$, with $m = \mathbf{A} + \mathbf{D}$ results in many identical integers t_m. Removing these duplicate integers can be achieved by first generating and storing all the resulting integers and subsequently sorting and uniqueing this large dataset. To avoid storing all the resulting integers for a given pair $(\mathbf{A},\mathbf{D})$, which requires a significant amount of storage as we will see later, and to avoid sorting this huge data set we define a more restricted set of rules $\mathcal{Q}_m \subset \mathcal{P}_m \subset O_m$ as follows

$$\begin{aligned} \mathcal{Q}_m = \big\{(o_{m-1},\ldots,o_0) \in \mathcal{P}_m \mid\ & o_0 = D, o_{m-1} \in \{A_i,S_i\}, 0 < k < m-1 : o_k \in \{D\} \cup \{A_i,S_i\}, \\ & o_k \in \{A_i,S_i\} \Rightarrow o_{k-1} = D \wedge (i = 0 \ \vee\ o_{i-1} \in \{A_\ell,S_\ell\})\big\}. \end{aligned}$$

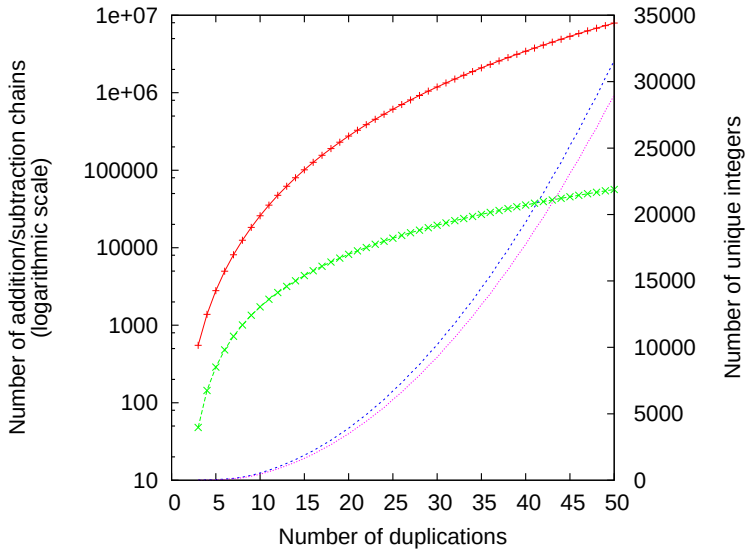

Fig. 1. The two top lines on the left denote the number of generated addition chains computing odd integers with $\mathcal{P}_m$ (upper (red) line) and Q_m (lower (green) line) when fixing **A**=3 and varying the number of duplications from one to fifty. The lower two lines show the number of *unique* integers corresponding to these chains where the upper line corresponds to $\mathcal{P}_m$.

The restrictions used in the definition of Q_m ensure that the resulting integer is odd and only addition (or subtraction) of an odd number to the current (even) number is allowed. This approach significantly reduces the amount of chains which produce the same resulting integer at the cost of slightly reducing the number of unique integers produced. To illustrate, Figure 1 shows the number of tuples generated by $\mathcal{P}_m$ and Q_m when using $\mathbf{A} = 3$ additions and $3 \leq \mathbf{D} \leq 50$ duplications resulting in odd integers. For $\mathbf{D} = 50$ the total number of tuples generated by $\mathcal{P}_{53}$ is more than 140 times higher compared to Q_{53} while the number of unique odd resulting integers is only 1.09 times higher.

The list of $m+1$ integers u_i corresponding to the m-tuple of symbols from Q_m can be efficiently generated recursively using

$$u_{i+1} = \begin{cases} 2u_i \\ u_i \pm u_j \text{ for } j < i \text{ and } 2 \mid u_i, 2 \nmid u_j \end{cases}$$

with $u_0 = 1$ and ensuring that the final operation is not a duplication (to make the resulting integer odd). Hence, the next integer in the sequence can always be obtained by duplication or adding a previous odd number u_j to the current even integer u_i. The number of times a different u_j is used for addition determines the required amount of storage needed. In practice we generate all sequences using a fixed number of duplications **D** and additions **A** making sure that the resulting storage requirement is never too large.

A sequence of additions and duplications corresponding to the chains resulting from Q_m looks like

$$A_{i_{\mathbf{A}-1}} D^{d_{\mathbf{A}-1}} \ldots A_{i_1} D^{d_1} A_{i_0} D^{d_0} = (A_{i_{\mathbf{A}-1}} D) D^{d_{\mathbf{A}-1}-1} \ldots (A_{i_1} D) D^{d_1-1} (A_{i_0} D) D^{d_0-1} \quad (1)$$

with $\mathbf{D} = \sum_{i=0}^{\mathbf{A}-1} d_i$, $d_i > 0$, and indices i_j that satisfy the restrictions of Q_m, i.e., i_j takes one of the values $\sum_{g=1}^{h} (d_g + 1)$ for $0 \leq h \leq j$. Such a sequence starts with a duplication, ends with an addition

and an addition is always preceded by a duplication. Hence, there are $\binom{\mathbf{D}-1}{\mathbf{A}-1}$ choices for the order of the $\mathbf{A}-1$ pairs $(A_{i_j}D)$ and the $\mathbf{D}-\mathbf{A}$ duplications D. Since every addition can be substituted by a subtraction the number of possibilities is multiplied by a factor $2^{\mathbf{A}}$. The indices i_j can be chosen in $\mathbf{A}!$ ways, whence the total number of resulting integers produced by Q_m is

$$\binom{\mathbf{D}-1}{\mathbf{A}-1} \cdot \mathbf{A}! \cdot 2^{\mathbf{A}} = 2^{\mathbf{A}} \cdot \mathbf{A} \cdot \prod_{i=1}^{\mathbf{A}-1} (\mathbf{D}-\mathbf{A}+i).$$

The No-Storage Setting. The second setting we consider is constructing chains which do not require any additional stored points, besides the in- and output (and possibly some auxiliary variables required to calculate the elliptic curve group operation). This means we are looking for integers which can be computed using addition chains which only use duplications and add or subtract the input point. We can define the set of tuples $\mathcal{R}_m \subset Q_m$ as $\mathcal{R}_m = \{(o_{m-1},\ldots,o_0) \in Q_m \mid o_k \in \{A_0,S_0,D\}, 0 \le k < m\}$. All no-storage chains which can be constructed using $\mathbf{A}$ elliptic curve additions and $\mathbf{D}$ elliptic curve duplications are of the form

$$2^{\mathbf{D}} + \sum_{i=0}^{\mathbf{A}-1} \pm 2^{n_i}, \quad \text{with } 0 = n_0 < n_1 < \ldots < n_i < \ldots < n_{\mathbf{A}-1} < \mathbf{D}.$$

This form follows from (1) by setting $i_j = 0$; we have $n_i = \mathbf{D} - \sum_{g=1}^{i} d_{\mathbf{A}-g}$. Using the same argument as in the low-storage setting the number of resulting integers generated by $\mathcal{R}_m$ is $\binom{\mathbf{D}-1}{\mathbf{A}-1} \cdot 2^{\mathbf{A}}$. Compared to the low-storage setting the number is reduced by a factor of $\mathbf{A}!$, reflecting the missing choice of the indices i_j.

4 Combining Addition Chains

Recall that, given a bound B_1, we want to perform an elliptic curve scalar multiplication with the integer $k = \prod_{i=1}^{\ell} p_i = \mathrm{lcm}(1,\ldots,B_1)$ where the product ranges over ℓ (not necessarily distinct) primes. We can get rid of the problems posed by the primes 2 in this product by noticing that they can be handled by a sequence of duplications at the end of the ECSM and assuming in the following that all s_i are odd. The techniques from the previous section provide us with a lot of integers which can be constructed using a known number of additions (here we count subtractions as additions) and duplications. Since different addition chains can lead to the same integer we pick for each of these integers one addition chain (preferably the one with the lowest cost). In this way we get a list of distinct integers, each with an associated addition chain. We index this list by an index set I and call s_i the integer corresponding to $i \in I$. For $i \in I$ denote by $\mathrm{add}(s_i)$ resp. $\mathrm{dup}(s_i)$ the number of additions resp. duplications in the addition chain and by $\{s_{i,1},\ldots,s_{i,t_i}\}$ the multiset of the primes in the prime decomposition of s_i. Furthermore, let $\mathrm{cost}(s_i)$ be the cost of performing a scalar multiplication with s_i using the associated addition chain. A reasonable choice for Edwards curves is $\mathrm{cost}(s_i) = 7\mathrm{dup}(s_i) + 8\mathrm{add}(s_i) + 1$ which will be discussed in the next section.

Ideally, we want to find a subset $I' \subset I$ such that $k \mid \prod_{i\in I'} s_i$ and $\sum_{i\in I'} \mathrm{cost}(s_i)$ is minimal. To facilitate our task we will modify this in two ways. If the product in the first condition is bigger than k we do more work than necessary. This can lead to a lower cost, but we assume that replacing the first condition by $k = \prod_{i\in I'} s_i$ will not increase the minimum of $\sum_{i\in I'} \mathrm{cost}(s_i)$ significantly. The second modification is the replacement of $\sum_{i\in I'} \mathrm{cost}(s_i)$ by $\sum_{i\in I'} \mathrm{add}(s_i)$. To explain why we think that this does not increase the minimum too much we consider subsets I' for which $\sum_{i\in I'} \mathrm{cost}(s_i)$ is close to the

minimum. Then most s_i have a high ratio $\frac{\mathrm{dup}(s_i)}{\mathrm{add}(s_i)}$ and therefore we have for most of them $s_i \approx 2^{\mathrm{dup}(s_i)}$. Since $\prod_{i\in I'} s_i = k$ the sum $\sum_{i\in I'} \mathrm{dup}(s_i) \approx \log_2(k)$ does not vary too much. Furthermore, the summand 1 in the cost function is the least significant term and the cardinality of I' does not vary much. We are aware that the second modification is more delicate than the first one, but, as explained below, we will generate many sets I' and will pick the best one amongst them using the more costly function $\mathrm{cost}(s_i)$.

The condition $k = \prod_{i\in I'} s_i$ implies that every s_i in this product is B_1-powersmooth which suggests the following two stage approach:

1. Restrict to $\hat{I} = \{i \in I \mid s_i \text{ is } B_1\text{-powersmooth}\}$.
2. Find a subset $I' \subset \hat{I}$ such that the multisets

$$\bigcup_{i\in I'} \{s_{i,1}, \ldots, s_{i,t_i}\} = \{p_1, \ldots, p_\ell\}$$

coincide and that $\sum_{i\in I'} \mathrm{add}(s_i)$ is minimal.

Testing a large list of numbers for B_1-powersmoothness can be done using the method from [12, Section 4]. The main idea is to build a product tree from the list, replace the root node R (the product of all numbers of the list) by $k \bmod R$ (where $k = \mathrm{lcm}(1, \ldots, B_1)$ is precomputed) and then tree-wise replace each node by the residue of k modulo the node. The leaves resulting in 0 contained B_1-powersmooth numbers and their factorizations can be obtained by other means.

Finding an optimal set I' is in general a difficult problem and has been studied in [27]. We choose to use a greedy approach which results in satisfactory results. We start with an empty set I' and the multiset $M = \{p_1, \ldots, p_\ell\}$ of primes to be matched. As long as M is non-empty we select an integer $s_i = \prod_{j=1}^{t_i} s_{i,j}$ with $\{s_{i,1}, \ldots s_{i,t_i}\} \subset M$ such that the ratio $\frac{\mathrm{dup}(s_i)}{\mathrm{add}(s_i)}$ is high and replace I' by $I' \cup \{i\}$ and M by $M \setminus \{s_{i,1}, \ldots s_{i,t_i}\}$.

This may fail because we might not be able to satisfy the condition $\{s_{i,1}, \ldots s_{i,t_i}\} \subset M$ at a given point. There are several ways to overcome this problem, e.g., we could increase our supply of s_i by generating more addition chains. Another way consists in aborting the greedy search at this point, getting $k = c \cdot \prod_{i\in I'} s_i$ for some integer c. Using the method of Dixon/Lenstra, we can search for a decomposition of c into several factors, each having a good addition chain. For the sizes of B_1 considered in this paper, namely $B_1 \leq 8192$, c consisted of very few primes and was often 1. Therefore the usually lower duplication/addition ratio of the c-part does not pose a problem for small B_1.

A refinement to this approach is to also take the size of the prime factors $s_{i,j}$ into account. A strategy could be to prefer choosing integers s_i which have mostly large prime divisors, since the majority of the primes p_i is large. The idea is to attach a score to a B_1-powersmooth integer given its prime factorization with respect to the currently unmatched prime factors in k. For a multiset N of primes bounded by B_1 the ratio of j-bit primes is defined as

$$a_j(N) := \frac{\#\{p \in N \mid \lceil \log_2(p) \rceil = j\}}{\#M},$$

where $1 \leq j \leq \lceil \log_2(B_1) \rceil$. Given M, the multiset of currently unmatched primes, the *score* of s_i is defined as

$$\mathrm{score}\left(s_i = \prod_{j=1}^{t_i} s_{i,j}, M\right) = \sum_{\substack{h=1: \\ a_h(M)\neq 0}}^{\lceil \log_2(B_1) \rceil} \frac{a_h(s_{i,1}, \ldots, s_{i,t_i})}{a_h(M)}$$

The higher the score the more small prime divisors are likely to be present. In general, for a given ratio, we select the integers which have a low score.

To illustrate, consider $B_1 = 1024$ where the initial a_i are

Algorithm 1 Given a bound B_1 and a set of B_1-powersmooth integers $\{s_i \mid i \in \hat{I}\}$, which can be computed with an addition chain using $\mathrm{add}(s_i)$ resp. $\mathrm{dup}(s_i)$ elliptic curve additions resp. duplications, together with the prime factorization of these integers ($s_i = \prod_j s_{i,j}$) the algorithm attempts to output triples $(s_j, \mathrm{add}(s_j), \mathrm{dup}(s_j))$ such that $\mathrm{lcm}(1,\ldots,B_1) = c \cdot \prod_j s_j$ for a small integer c. This algorithm considers scores $\leq s_{\mathrm{thres}}$ only and combines integers s_i for which $\dfrac{\mathrm{dup}(s_i)}{\mathrm{add}(s_i)} \geq r$ where r starts at r_h and is decreased until r_l.

Input:
- Bound $B_1 \in \mathbf{Z}$, we have $\mathrm{lcm}(1,\ldots,B_1) = \prod_{i=1}^{\ell} p_i$ with p_i prime,
- Set of integers $\{s_i \mid i \in \hat{I}\}$ with $s_i = \prod_j s_{i,j}$ for $s_{i,j}$ prime and $i \in \hat{I}$,
- Upper and lower bound on the duplication/addition ratio: r_h and r_l
- A threshold value for the score: T

Output: Triples $(s_i, \mathrm{add}(s_i), \mathrm{dup}(s_i))$ and c such that $c \cdot \prod_i s_i = \mathrm{lcm}(1,\ldots,B_1)$

1. $M \leftarrow \{p_1,\ldots,p_\ell\}$, $I' \leftarrow \emptyset$
2. **for** $r = r_h$ to r_l **do**
3. found $\leftarrow$ true
4. **while** found=true **do**
5. found $\leftarrow$ false, $j \leftarrow 1$
6. **for** $i \in \hat{I}$ **do**
7. **if** $\{s_{i,1},\ldots s_{i,t_i}\} \subset M$ and $\dfrac{\mathrm{dup}(s_i)}{\mathrm{add}(s_i)} \geq r$ and $\mathrm{score}(s_i, M) \leq T$ **then**
8. j++, $\mathrm{score}_j \leftarrow (\mathrm{score}(s_i, M), i)$
9. sort score_i for $1 \leq i \leq j$ with respect to $\mathrm{score}(s_i, M)$
10. **if** $j \geq 1$ **then**
11. $i \leftarrow$ index from score_1, output $(s_i, \mathrm{add}(s_i), \mathrm{dup}(s_i))$
12. $I' \leftarrow I' \cup \{i\}$, $M \leftarrow M \setminus \{s_{i,1},\ldots s_{i,t_i}\}$ /* Update I' and M */
13. found $\leftarrow$ true
14. output $c = \prod_{p \in M} p$

$$a_2 = 0.032,\ a_3 = 0.037,\ a_4 = 0.021,\ a_5 = 0.053,\ a_6 = 0.037,$$
$$a_7 = 0.069,\ a_8 = 0.122,\ a_9 = 0.229,\ a_{10} = 0.399$$

(with $\sum_{i=2}^{10} a_i = 1$). Almost 40 percent of all the primes fall in the largest (10-bit) category. An example of a low score-integer is $11529215054666795009 = 743 \cdot 719 \cdot 677 \cdot 461 \cdot 457 \cdot 449 \cdot 337$ where the size of the smallest prime is 9-bit, the score is 3.57 and this integer can be computed using 63 duplications and five additions as $A_0 D^{11} A_0 D^{12} A_0 D^{10} A_0 D^{28} A_0 D^2 \in \mathcal{R}_{68}$. On the other hand, an example of a high-score integer, consisting of mainly small primes, is $1048575 = 41 \cdot 31 \cdot 11 \cdot 5^2 \cdot 3$, its score is significant higher (29.62) and it can be computed with 20 duplications and a single subtraction as $S_0 D^{20} \in \mathcal{R}_{21}$.

This approach using scores is outlined in Algorithm 1. Note that the scores are recalculated each time an s_i is chosen. In practice one could reduce the amount of these costly recalculations by picking several s_i in lines 10-13 of the algorithm; in this case one has to check that the union of the prime factors of the chosen s_i is still a multisubset of M.

A Randomized Variant In the current state, Algorithm 1 returns a single solution given a set of input parameters. To increase the amount of different subsets I', and hereby hopefully improving the results, we randomize the selection process of the index that is added in lines 10-13 of the algorithm. With probability $x \in \mathbf{R}$ $(0 < x < 1)$ select the s_i corresponding to score_1 or, with probability $1 - x$, skip it and repeat this procedure for score_2 and so on. If we have reached the end of the list (after j trials) one could apply a deterministic choice.

Table 2. The left table shows the number of integers generated with an addition chain using **A** and **D** elliptic curve additions and duplications respectively. All these integers were tested for $2.9 \cdot 10^9$-powersmoothness and, if smooth, the prime divisors are stored. The **bold** ranges indicate that 2^{31} random integers per single **A**, **D** combination were tested for smoothness instead of the full range. The right table shows the number of unique B_1-powersmooth integers in the no-storage and low-storage setting for different values of B_1.

No-storage setting			Low-storage setting		
A	**D**	#smoothness tests	**A**	**D**	#smoothness tests
1	$5-200$	$3.920 \cdot 10^2$	1	$5-250$	$4.920 \cdot 10^2$
2	$10-200$	$7.946 \cdot 10^4$	2	$10-250$	$2.487 \cdot 10^5$
3	$15-200$	$1.050 \cdot 10^7$	3	$15-250$	$1.235 \cdot 10^8$
4	$20-200$	$1.035 \cdot 10^9$	4	$20-250$	$6.101 \cdot 10^{10}$
5	$25-200$	$8.114 \cdot 10^{10}$	5	$25-153$	$2.511 \cdot 10^{12}$
			5	**$154-220$**	$\mathbf{1.439 \cdot 10^{11}}$
6	$30-124$	$2.858 \cdot 10^{11}$	**6**	**$60-176$**	$\mathbf{2.513 \cdot 10^{11}}$
7	$35-55$	$2.529 \cdot 10^{10}$			
Total		$3.932 \cdot 10^{11}$			$2.967 \cdot 10^{12}$

B_1	No-Storage	Low-Storage
256	$2.412 \cdot 10^5$	$9.012 \cdot 10^6$
512	$1.442 \cdot 10^6$	$3.013 \cdot 10^7$
1024	$5.466 \cdot 10^6$	$7.271 \cdot 10^7$
8092	$8.034 \cdot 10^7$	$4.399 \cdot 10^8$
$2.9 \cdot 10^9$	$1.054 \cdot 10^{10}$	$3.930 \cdot 10^{10}$

5 Additional Multiplications

The fastest arithmetic for Edwards curves is due to Hisil et al. [16]. They propose to use extended twisted Edwards coordinates, which are twisted Edwards coordinates plus an auxiliary coordinate. This allows faster addition but slower duplication. Using a mixing technique, by switching between extended twisted Edwards and regular twisted Edwards, the overall cost for scalar multiplication is reduced [16]. This is realized by performing the duplications using the cheaper regular twisted Edwards coordinates when a duplication is followed by a duplication. When an addition is required after a duplication one can use the duplication formula in the extended twisted Edwards coordinates (which does not need the auxiliary coordinate as input) at the cost of an extra multiplication to compute the auxiliary coordinate of the result. Next, the fast addition is performed in extended twisted Edwards coordinates; one multiplication (to compute the auxiliary coordinate of the output) can be saved, cancelling the extra multiplication used when doubling, since a duplication is always performed after an addition in ECSM-algorithms. This approach assumes that both inputs of the elliptic curve addition are in extended twisted Edwards coordinates. This is the case for double-and-add algorithms and (signed) windowing algorithms where the computation of the auxiliary coordinates of the lookup table are a minor overhead.

In both our settings, where we consider low- and no-storage, this does not hold. Converting a point from twisted Edwards coordinates to extended twisted Edwards coordinates requires a single multiplication. The computation of the large elliptic curve scalar product is done by processing batches of prime products (the s_i) at a time. All the additions or subtractions required in the addition chain to compute s_i require that the points are in extended twisted Edwards coordinates. When required, the odd intermediate results are stored in extended twisted Edwards coordinates at a cost of a single additional multiplication. The cost of computing a low-storage addition chain $(o_{m-1}, \ldots, o_0) \in Q_m$ resulting in s_i is increased by $x(s_i)$ multiplications, where $x(s_i) = \#\{j \mid \exists h : o_h \in \{A_j, S_j\}, 0 \leq h < m\}$; i.e. the unique number of indices used in the additions and subtractions. Therefore we get for the cost function from the previous section $\text{cost}(s_i) = 7\text{dup}(s_i) + 8\text{add}(s_i) + x(s_i)$. In the no-storage setting we always have $x(s_i) = 1$ leading to the choice for $\text{cost}(s_i)$ given at the beginning of the previous section. In total we have #{addition chains used} additional multiplications in the no-storage setting and a potentially higher number in the low-storage setting. We can save one multiplication due to the

sequence containing the power of 2 (which consists of duplications only) and another multiplication if we assume that the input point is already in extended twisted Edwards coordinates.

6 Results

Using the rules given in section 3.2 for both the no-storage and the low-storage setting, we generated more than 10^{12} integers for many choices of the number of additions **A** and duplications **D**. Table 2 summarizes the ranges we have covered where bold ranges (in the low-storage setting) indicate that only 2^{31} random integers were generated instead of the full range. All these integers were subjected to $2.9 \cdot 10^9$-powersmoothness tests which reduced the number of integers by about two orders of magnitude. This large powersmoothness-bound was chosen to facilitate searching for efficient addition chains for much larger B_1 parameters, which is the subject of a separate project and can be used for factoring larger numbers (see e.g. [6]). From the reduced set of integers we extracted those that are B_1-powersmooth for the values of B_1 used in this paper (see right part in table 2). These computations were done on five 8-core Intel Xeon E5430 (2.66GHz) and took more than half a year. The smoothness testing required most of the run-time and up to 4.6GB of memory. Using the approach outlined in Algorithm 1 one of these nodes was occasionally used for the combining experiments which consisted of thousands of runs of the randomized greedy approach, each of them taking only a couple of seconds for these low values of B_1.

Table 4 in Appendix A shows an example for $B_1 = 256$ in the no-storage setting. All the prime powers $p^e \leq 256$ with p prime, $e \in \mathbf{Z}$ such that $p^{e+1} > 256$ are used (using exactly the same prime powers as in GMP-ECM and EECM-MPFQ). The total cost, in terms of modular multiplications and squarings, for these 15 addition chains is $361 \times (3\mathbf{M} + 4\mathbf{S}) + 38 \times 8\mathbf{M} + 13\mathbf{M} = 1\,444\mathbf{S} + 1\,400\mathbf{M}$ where the 13 additional multiplications are due to changing to extended twisted Edwards coordinates in all except the first and last chain (row) in Table 4. Only additions or subtractions with the input point are performed, hence no storage besides the in- and output is required.

Table 3 shows the results obtained using Algorithm 1 on our dataset (see Table 2). The memory required is expressed in the number of residues (R), integers modulo n, which need to be kept in memory. Here we assume that extended twisted Edwards coordinates are used, i.e., every point is represented by four coordinates. In the setting of EECM-MPFQ [2, 1] we assume that an optimal window size is used and that besides the window table only the input point needs to be kept in memory while we assume that two points (the input point and the current active point) are required in the no- and low-storage setting. The implementation of the elliptic curve group operation is assumed to require at most two auxiliary variables (residues). Hence, the no-storage setting requires memory for $2 \times 4 + 2 = 10$ residues modulo n.

The low-storage results presented in Table 3 require to store at most three additional points (12 residues modulo n). This more than doubles the amount of storage required when using the no-storage setting but is still significantly less compared to the approach used in [2, 1].

6.1 Application to GPUs

When running ECM on memory constrained devices, like GPUs, the large number of precomputed points required for the windowing methods cannot be stored. Typically one is forced to settle for a (much) smaller window size reducing the advantage from using twisted Edwards curves. For example, in [4] no large window sizes are used at all, the authors remark: “Besides the base point, we cannot cache any other points”. Memory is also a problem in [3], the faster curve arithmetic from Hisil et al. [16] is not used since this requires storing a fourth coordinate per point.

Table 3. The number of modular multiplications (**M**) and squarings (**S**) required to calculate **A** elliptic curve additions and **D** duplications for various B_1 parameters when factoring an integer n with ECM. The memory required is expressed as the number of residues (R), integers modulo n, which are kept in memory. The performance speedup (in terms of #**M** + #**S**) and memory reduction compared to the Edwards ECM approach from [1] is given.

B_1	#**M**	#**S**	#**M** + #**S**	speedup	**A**	**D**	#R	reduction
256 [1]	1 638	1 436	3 074		69	359	38	
No storage	1 400	1 444	2 844	1.08	38	361	10	3.80
Low storage	1 383	1 448	2 831	1.09	35	362	22	1.73
512 [1]	3 183	2 952	6 135		120	738	62	
No storage	2 842	2 964	5 806	1.06	75	741	10	6.20
Low storage	2 776	2 964	5 740	1.07	65	741	22	2.82
1 024 [1]	6 144	5 892	12 036		215	1 473	134	
No storage	5 596	5 912	11 508	1.05	141	1 478	10	13.40
Low storage	5 471	5 904	11 375	1.06	123	1 476	22	6.09
8 192 [1]	45 884	47 156	93 040		1 314	11 789	550	
No storage	43 914	47 160	91 074	1.02	1 043	11 790	10	55.00
Low storage	42 855	47 136	89 991	1.03	878	11 784	22	25.00

From the data given in Table 3 it becomes clear that our approach reduces the memory requirements significantly. For example, the memory required to run ECM in the cofactorization setting on GPUs considered in [4, 3] (using $B_1 = 8\,192$) can be reduced by a factor 55 when using the addition chains from our no-storage setting. Hence, when using the methods described in this paper *less* memory is required allowing the usage of the *faster* curve arithmetic and *reducing* the number of elliptic curve additions required in the computation of the elliptic curve scalar multiplication. When running Edwards ECM on GPUs in practice one of the main bottlenecks is the limited memory available. Therefore by lowering the memory footprint one expects to get an additional speedup which might be even larger than the speedup reported in Table 3.

7 Conclusions

The relatively new Edwards curves combined with the fast arithmetic from extended twisted Edwards coordinates are faster compared to using Montgomery curves. This speed-up comes at a price, namely a larger memory requirement which, when optimizing for speed, grows roughly linearly in the size of B_1, whereas the memory requirement in the Montgomery curves setting is constant and small. Inspired by the approach from Dixon and Lenstra and using the fact that only a few popular B_1-values are used in practice in NFS, we have presented techniques to reduce the memory requirement significantly by doing precomputations for these B_1-values. In these precomputations we tested over 10^{12} integers coming from additions chains with a low addition/duplication ratio for smoothness and combined them using a greedy approach. Our results show that we require significantly less memory compared to the current state-of-the-art Edwards ECM approach, and are even slightly faster. This makes our approach extremely suitable for memory-constrained parallel architectures like GPUs.

References

1. D. J. Bernstein, P. Birkner, and T. Lange. Starfish on strike. In M. Abdalla and P. S. L. M. Barreto, editors, *Latincrypt*, volume 6212 of *Lecture Notes in Computer Science*, pages 61–80. Springer, Heidelberg, 2010.
2. D. J. Bernstein, P. Birkner, T. Lange, and C. Peters. ECM using Edwards curves. Cryptology ePrint Archive, Report 2008/016, 2008. `http://eprint.iacr.org/`.

3. D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang. The billion-mulmod-per-second PC. In *Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2009*, pages 131–144, 2009.
4. D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on graphics cards. In A. Joux, editor, *Eurocrypt 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 483–501. Springer, Heidelberg, 2009.
5. D. J. Bernstein and T. Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. In G. L. Mullen, D. Panario, and I. E. Shparlinski, editors, *Finite Fields and Applications*, volume 461 of *Contemporary Mathematics Series*, pages 1–19. American Mathematical Society, 2008.
6. J. W. Bos, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Efficient SIMD arithmetic modulo a Mersenne number. In *IEEE Symposium on Computer Arithmetic – ARITH-20*, pages 213–221. IEEE Computer Society, 2011.
7. A. Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45:736–739, 1939.
8. R. P. Brent. Some integer factorization algorithms using elliptic curves. *Australian Computer Science Communications*, 8:149–163, 1986.
9. G. de Meulenaer, F. Gosset, G. M. de Dormale, and J.-J. Quisquater. Integer factorization based on elliptic curve method: Towards better exploitation of reconfigurable hardware. In *Field-Programmable Custom Computing Machines – FCCM 2007*, pages 197–206. IEEE Computer Society, 2007.
10. B. Dixon and A. K. Lenstra. Massively parallel elliptic curve factoring. In R. A. Rueppel, editor, *Eurocrypt 1992*, volume 658 of *Lecture Notes in Computer Science*, pages 183–193. Springer, Heidelberg, 1993.
11. H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, July 2007.
12. J. Franke, T. Kleinjung, F. Morain, and T. Wirth. Proving the primality of very large numbers with fastECPP. In D. A. Buell, editor, *Algorithmic Number Theory – ANTS-VI*, volume 3076 of *Lecture Notes in Computer Science*, pages 194–207. Springer, Heidelberg, 2004.
13. K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi. Implementing the elliptic curve method of factoring in reconfigurable hardware. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 119–133. Springer, Heidelberg, 2006.
14. T. Güneysu, T. Kasper, M. Novotny, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57:1498–1513, 2008.
15. R. Guy. *Unsolved problems in number theory*, volume 1. Springer Verlag, 3rd edition, 2004.
16. H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In J. Pieprzyk, editor, *Asiacrypt 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer, Heidelberg, 2008.
17. T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In T. Rabin, editor, *Crypto 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, Heidelberg, 2010.
18. A. Kruppa. A software implementation of ECM for NFS. Research Report RR-7041, INRIA, 2009. `http://hal.inria.fr/inria-00419094/PDF/RR-7041.pdf`.
19. A. K. Lenstra and H. W. Lenstra, Jr. Algorithms in number theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Volume A: Algorithms and Complexity)*, pages 673–715. Elsevier and MIT Press, 1990.
20. A. K. Lenstra and H. W. Lenstra, Jr. *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verslag, 1993.
21. H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.
22. D. Loebenberger and J. Putzka. Optimization strategies for hardware-based cofactorization. In M. J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 170–181. Springer, Heidelberg, 2009.
23. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
24. P. L. Montgomery. *An FFT extension of the elliptic curve method of factorization*. PhD thesis, University of California, 1992.
25. NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi, 2009.
26. J. Pelzl, M. Šimka, T. Kleinjung, M. Drutarovský, V. Fischer, and C. Paar. Area-time efficient hardware architecture for factoring integers with the elliptic curve method. *Information Security, IEE Proceedings on*, 152(1):67–78, 2005.
27. D. Pisinger. A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 83(2):394–410, June 1995.
28. J. M. Pollard. The lattice sieve. pages 43–49 in [20].
29. J. M. Pollard. Theorems on factorization and primality testing. *Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974.

30. C. Pomerance. The quadratic sieve factoring algorithm. In T. Beth, N. Cot, and I. Ingemarsson, editors, *Eurocrypt 1984*, volume 209 of *Lecture Notes in Computer Science*, pages 169–182. Springer, Heidelberg, 1985.
31. A. Scholz. Aufgabe 253. *Jahresbericht der deutschen Mathematiker-Vereingung*, 47:41–42, 1937.
32. J. H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Gradute Texts in Mathematics*. Springer-Verlag, 1986.
33. M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, and V. Fischer. Hardware factorization based on elliptic curve method. In *Field-Programmable Custom Computing Machines – FCCM 2005*, pages 107–116. IEEE Computer Society, 2005.
34. E. G. Thurber. On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal*, 40:907–913, 1973.
35. P. Zimmermann and B. Dodson. 20 years of ECM. In F. Hess, S. Pauli, and M. E. Pohst, editors, *Algorithmic Number Theory – ANTS-VII*, volume 4076 of *Lecture Notes in Computer Science*, pages 525–542. Springer, Heidelberg, 2006.
36. R. Zimmermann, T. Güneysu, and C. Paar. High-performance integer factoring with reconfigurable devices. In *Field Programmable Logic and Applications – FPL 2010*, pages 83–88. IEEE, 2010.
37. P. Zimmermann et al. GMP-ECM (elliptic curve method for integer factorization). `https://gforge.inria.fr/projects/ecm/`, 2012.

A Appendix

Table 4. Example of the best addition chain found for $B1 = 256$ in the no-storage setting.

#D	#A	product	addition chain
11	1	$89 \cdot 23$	S_0D^{11}
14	2	$197 \cdot 83$	$S_0D^5S_0D^9$
15	2	$193 \cdot 191$	$S_0D^{12}A_0D^3$
15	2	$199 \cdot 19 \cdot 13$	$A_0D^{14}A_0D^1$
18	1	$109 \cdot 37 \cdot 13 \cdot 5$	A_0D^{18}
19	2	$157 \cdot 53 \cdot 7 \cdot 3 \cdot 3$	$S_0D^6S_0D^{13}$
21	3	$223 \cdot 137 \cdot 103$	$A_0D^{10}A_0D^{10}A_0D^1$
23	3	$179 \cdot 149 \cdot 61 \cdot 5$	$S_0D^{13}A_0D^5S_0D^5$
28	1	$127 \cdot 113 \cdot 43 \cdot 29 \cdot 5 \cdot 3$	S_0D^{28}
30	3	$181 \cdot 173 \cdot 167 \cdot 11 \cdot 7 \cdot 3$	$A_0D^{11}A_0D^{16}A_0D^3$
33	5	$211 \cdot 73 \cdot 67 \cdot 59 \cdot 47 \cdot 3$	$S_0D^6A_0D^2A_0D^{11}S_0D^3S_0D^{11}$
36	4	$241 \cdot 131 \cdot 101 \cdot 79 \cdot 31 \cdot 11$	$A_0D^2A_0D^{16}A_0D^{16}A_0D^2$
41	4	$233 \cdot 229 \cdot 163 \cdot 139 \cdot 107 \cdot 17$	$S_0D^9S_0D^4S_0D^{11}S_0D^{17}$
49	5	$251 \cdot 239 \cdot 227 \cdot 151 \cdot 97 \cdot 71 \cdot 41$	$S_0D^3S_0D^{29}A_0D^4A_0D^8A_0D^5$
8	0	2^8	D^8
361	38	**Total**	