


Montgomery inversion

Erkay Savaş¹  · Çetin Kaya Koç²

Received: 4 September 2016 / Accepted: 22 March 2017 / Published online: 18 April 2017
© Springer-Verlag Berlin Heidelberg 2017

Abstract Multiplicative inversion in finite fields is an essential operation in many cryptographic applications such as elliptic curve and pairing-based cryptography. While the classical extended Euclidean algorithm involves expensive division operations, the binary extended Euclidean and Kaliski's algorithms use simple shift, addition and subtraction operations. The Montgomery inverse operation is applied when the Montgomery multiplication operation is used for fast arithmetic. As the inversion operation is applied to sensitive data, a constant-time inversion algorithm is useful against a class of side-channel attacks. In this paper, we show different ways of computing the Montgomery inverse of a given integer and compare their complexity in terms of the number of additions/subtractions and shift operations. We also propose a simple parallel algorithm to compute Montgomery inverse, which can be useful in multi-core processors where data sharing among cores is relatively inexpensive. Finally, we propose two efficient constant-time Montgomery inversion algorithms, which are useful as countermeasures against side-channel attacks.

Keywords Multiplicative inverse · Extended Euclidean algorithm · Montgomery inverse · Constant-time implementation · Parallelization

1 Introduction

Multiplication and inversion operations in rings and finite fields are the most frequently used arithmetic operations in many cryptographic algorithms such as RSA algorithm [1], Diffie–Hellman key exchange algorithm [2], the US federal Digital Signature Standard (DSS) [3], elliptic and hyperelliptic curve cryptography [4–6], and pairing-based cryptography [7]. The multiplication operation dominates the execution times of these algorithms, and thus, its performance is of extreme importance. The Montgomery multiplication algorithm introduced in a seminal work by Montgomery [8] is usually considered to be the most efficient method in digital computers as it replaces prohibitively expensive division operations by bit-level or word-level shifts.

The Montgomery multiplication algorithm requires its operands in the Montgomery domain, i.e., $a2^n \bmod p$ and $b2^n \bmod p$, given $a, b < p$ where p is an odd integer or prime and n is generally taken as the bit size of the modulus p . The algorithm takes $a2^n \bmod p$ and $b2^n \bmod p$ as inputs and computes $c2^n \bmod p$, where $c = a \cdot b \bmod p$. In other words, the algorithm ensures the result in the Montgomery domain provided that the operands are given in the Montgomery domain. Therefore, other arithmetic operations are expected to satisfy the same condition when Montgomery arithmetic is used to avoid forward and backward transformation from and to the Montgomery domain. For addition and subtraction, the condition is satisfied immediately, as $a2^n \pm b2^n \equiv (a \pm b)2^n \equiv c2^n \bmod p$. The inversion, on the other hand, requires more effort since the classical inversion algorithm outputs $a^{-1}2^{-n} \bmod p$ given $a2^n \bmod p$, instead of $a^{-1}2^n \bmod p$. The correct definition of the Montgomery inversion would be $a^{-1}2^n \bmod p$, as given in [9].

The roots of direct inversion algorithms in finite fields can be traced to the Euclidean algorithm for the computa-

✉ Erkay Savaş
erkays@sabanciuniv.edu; erkay.savas@gmail.com

Çetin Kaya Koç
koc@cs.ucsb.edu

¹ Sabancı University, Istanbul, Turkey

² University of California Santa Barbara,
Santa Barbara, CA, USA

tion of greatest common divisor (GCD) reported by Euclid in his Elements [10]. Stein's algorithm [11] for computing the GCD of two nonnegative integers replaces division operations by less complex comparison, subtraction and shift operations. The extended version of Stein's algorithm [12], referred as the binary extended Euclidean algorithm (bEEA), can be used to compute the inverse of integer $a < p$, where $\text{GCD}(a, p) = 1$.

Kaliski's method [13] further improves bEEA by reducing the number of addition and subtraction operations. However, it computes $a^{-1}2^k \bmod p$ given $a < p$ (alternatively, $a^{-1}2^{k-n} \bmod p$ given $a2^n \bmod p$), where k is the number of iterations in the algorithm and follows a random distribution in the interval $[n, 2n]$. Therefore, Kaliski's method requires a correction phase to obtain the correct result. In the literature, several other algorithms are proposed to accelerate the computation of inversion operation in hardware or software implementations [14–19].

The inversion operation is used in elliptic curve cryptography to transform the projective coordinates to affine coordinates after the point operations are calculated in projective coordinates [20]. It is also used in DSS algorithm to compute the inverse of ephemeral key. Finally in elliptic curve cryptography, when precomputation is applied for scalar multiplication in the fixed-based point scenario, one can use the affine formulae for precomputed points, which requires the computation of inversion. In some cases, the computations involve sensitive information and simple side-channel attacks become feasible if the inversion computation is not protected. A recent work [21] modifies Kaliski's method for inversion to obtain a constant-time algorithm as a protection against a class of side-channel attacks.

Our contributions in this paper are as follows:

- We show that there are alternative methods to compute the Montgomery inverse of an integer modulo a prime integer including a method based on the classical extended Euclidean algorithm.
- We propose a simple method to parallelize the inversion algorithms, which can be useful for multi-core processors if inter-core data sharing is possible with low overhead.
- We provide two constant-time algorithms for computing Montgomery inversion efficiently.

Intentionally, we provide neither hardware nor software implementation results such as timing benchmarks on a specific platform as we are not proposing a new and fast algorithm for computing multiplicative inverse. Naturally, we refrain from giving any assessment as to whether inversion can be computed sufficiently fast to avoid using projective coordinates in elliptic curve cryptography, which eliminates almost all inversion operations at the expense of increased number of multiplications and higher mem-

ory/storage requirement. This paper can be regarded as a guide book or a short survey which can be useful for future work on the subject. The literature provides a wide collection of works on modular inversion targeting its various other aspects that are not included in this work. Hardware implementations of various designs for performing fast, efficient and scalable computation of inversion are reported in [22–29]. A recent work in [30] uses the approach based on Fermat's little theorem to compute multiplicative inverse in constrained computing platforms such as those used in wireless sensor nodes to save code space as the approach utilizes only modular multiplication as a building block. On the other hand, the work in [31] uses the extended Euclidean algorithm in an embedded processor. In [32], the authors discuss the ratio of inversion timing to multiplication timing which is the key factor in choosing between the affine and projective coordinates in elliptic curve cryptography. As the ratio is usually large as attested by state-of-the-art implementations such as GMP,¹ using projective coordinates that trades inversion for several extra multiplication operations is a better strategy of implementing elliptic curve cryptography. Finally, [21] proposes a constant-time inversion algorithm and reports implementation results.

The organization of the paper is as follows: in Sect. 2, we start with the classical Euclidean algorithm and show how we can modify it to compute the Montgomery inversion. Section 3 presents the definitions pertaining to Montgomery arithmetic. In Sect. 4, we describe the binary GCD algorithm in a slightly different way that is useful to follow the explanations in subsequent sections. Section 5 explains the method to compute the Montgomery inversion using bEEA while Sect. 6 focuses on Kaliski's inversion algorithm for the same purpose. Section 7 compares binary EEA and Kaliski's method from efficiency point of view. We briefly explain the parallelization idea that can be utilized with many inversion algorithms in Sect. 8. Constant-time inversion algorithms are provided in Sect. 9. Finally, we conclude our paper in Sect. 10.

2 The classical Euclidean algorithm

The Euclidean algorithm provides a simple means for computing the greatest common divisor, denoted $\text{GCD}(u, v)$ of two positive integers u and v without factoring them. The extended Euclidean algorithm (EEA) returns two unique integers s and r in addition to the greatest common divisor of u and v . The EEA is depicted in Algorithm 1. After the EEA successfully terminates, the following relation holds

$$ar + ps = \text{GCD}(a, p).$$

¹ GNU Multiple Precision Arithmetic Library: <http://www.gmp.org>.

Algorithm 1 Extended Euclidean Algorithm (EEA)

Input: a and p with $a < p$, where $\text{GCD}(a, p)=1$
Output: $r = a^{-1} \bmod p$
1: $u \leftarrow p, v \leftarrow a$
2: $s_2 \leftarrow 1, s_1 \leftarrow 0$
3: $r_2 \leftarrow 0, r_1 \leftarrow 1$
4: **while** $v > 0$ **do**
5: $Q \leftarrow \lfloor u/v \rfloor$
6: $R \leftarrow u - Q \cdot v$
7: $s \leftarrow s_2 - Q \cdot s_1$
8: $r \leftarrow r_2 - Q \cdot r_1$
9: $u \leftarrow v$
10: $v \leftarrow R$
11: $s_2 \leftarrow s_1$
12: $s_1 \leftarrow s$
13: $r_2 \leftarrow r_1$
14: $r_1 \leftarrow r$
15: **end while**
16: $r \leftarrow r_2$
17: $s \leftarrow s_2$
18: **return** r

If a and p are relatively prime, $\text{GCD}(a, p)$ would be equal to 1, and thus, it immediately follows that

$$ar \equiv 1 \pmod{p}.$$

Hence, the extended Euclidean algorithm computes the modular inverse $r = a^{-1} \bmod p$. The classical EEA can be modified to compute the Montgomery inverse of an integer by changing Step 3 of the algorithm as $r_1 \leftarrow 0$ and $r_1 \leftarrow 2^{2n} \bmod p$. Given the input $a < p$, the modified algorithm mEEA computes

$$r = \text{mEEA}(a2^n \bmod p, p) = a^{-1} \cdot 2^{-n} \cdot 2^{2n} \bmod p,$$

which is the Montgomery inverse of $a2^n \bmod p$. The classical inverse algorithm, on the other hand, heavily relies on division operation; although efficiently computed over integers with relatively small sizes and on computers with large word, division is a prohibitively expensive operation. Therefore, the algorithms in subsequent sections, which avoid expensive division operation, are preferred in many circumstances.

3 Definitions

In this section, we provide definitions pertaining to Montgomery arithmetic with focus on inversion operation. We always assume $p > 2$ is a prime throughout the paper.

Definition 1 The positive integer R with $\text{GCD}(p, R)=1$ is the *Montgomery constant*, and selected as a power of two for efficient arithmetic in digital computers. A typical selection is $R = 2^n \bmod p$, where $n = \lceil \log_2 p \rceil$; namely, n is the bit size of p . Without loss of generality, we always use this selection in the rest of the paper.

Definition 2 Given an integer $a \bmod p$, $a \cdot R \bmod p$ is said to be in the *Montgomery domain*.

Definition 3 Given two integers, $a, b < p$, the operation $a \cdot b \cdot R^{-1} \bmod p$ is the *Montgomery multiplication* of a and b and denoted as $\text{MonMul}(a, b, p)$. Given two integers in the Montgomery domain, $aR \bmod p$ and $bR \bmod p$, the Montgomery multiplication yields

$$\text{MonMul}(a, b, p) = aR \cdot bR \cdot R^{-1} = abR = cR \bmod p,$$

where $c = a \cdot b \bmod p$.

Definition 4 Given the positive integer in the Montgomery domain, $aR \bmod p$, the positive integer $r = a^{-1} \cdot R \bmod p$ is the *Montgomery inverse* of a .

Definition 5 Given the positive integer in the Montgomery domain, $a \cdot R \bmod p$, the integer $r = a^{-1} \bmod p$ is the *backward Montgomery inverse* of a .

Definition 6 Given the positive integer in the regular domain, $a \bmod p$, the integer $r = a^{-1}R \bmod p$, which is in the Montgomery domain, is the *forward Montgomery inverse* of a .

4 The binary GCD algorithm

Stein’s algorithm [11] is an efficient method for computing the greatest common divisor of two positive integers; it employs only comparison, subtraction and shift operations, which are easy to implement in digital computers. Its most important aspect is that expensive division operations in the classical Euclidean GCD algorithm are replaced by divisions by 2, which can be implemented as simple logical shift operations over the binary representation of the integers, hence the name binary GCD.

A version of Stein’s algorithm that works when the numbers are odd is depicted in Algorithm 2. Steps 3, 4 and 10 of the algorithm perform logical operations; therefore, Boolean variables π_i s take only binary values of TRUE and FALSE. Note that the while loop starts with a logical operation, which checks whether the Boolean variable π_0 is TRUE or FALSE. Note also that $\bar{\pi}_i$ denotes the complement of Boolean variable π_i . The operations in Steps 6–9 are predicated, which are executed only when the associated Boolean variable (henceforth predicate) in the bracket is TRUE. One important aspect of the notation is that all Boolean variables in Steps 6–9 are checked; it is as if there is an if statement preceding these steps as opposed to if-elseif-else statements. Note that only one of π_1 – π_4 is TRUE at a time; thus, only one of Steps 6–9 is executed in each iteration of the while loop. We adopt this notation for two basic reasons: (i) propose a method to parallelize the inversion algorithms

Algorithm 2 Binary GCD Algorithm (bGCD)

Input: a and p with $a < p$ where p is odd
Output: $u = \text{GCD}(a, p)$

- 1: $u \leftarrow p, v \leftarrow a$
- 2: $k \leftarrow 0$
- 3: $\pi_0 \leftarrow (v > 0)$
- 4: **while** π_0 **do**
- 5: $\pi_1 \leftarrow (u \text{ is even})$
 $\pi_2 \leftarrow (\bar{\pi}_1 \text{ and } (v \text{ is even})),$
 $\pi_3 \leftarrow (\bar{\pi}_1 \text{ and } \bar{\pi}_2 \text{ and } (u > v)),$
 $\pi_4 \leftarrow (\bar{\pi}_1 \text{ and } \bar{\pi}_2 \text{ and } \bar{\pi}_3),$
- 6: $[\pi_1]: u \leftarrow u/2$
- 7: $[\pi_2]: v \leftarrow v/2$
- 8: $[\pi_3]: u \leftarrow (u - v)/2$
- 9: $[\pi_4]: v \leftarrow (v - 2)/2$
- 10: $\pi_0 \leftarrow (v > 0)$
- 11: $k \leftarrow k + 1$
- 12: **end while**
- 13: **return** u

and (ii) explain versions of inversion algorithms that executes in constant time which protects them against a class of side-channel attacks.

From the complexity point of view, the expected number of iterations is experimentally found as about $1.4n$, where n is the bit size of p . In addition, we can give upper and lower bound for the number of iterations as $n \leq k \leq 2n$. For either u or v gets smaller after each iteration, performing shift and subtraction operations over these integers can become easier, depending on the implementation platform, as the computation progresses. In particular when multi-precision arithmetic is used in the implementation, it is advantageous to work with the exact lengths of u and v . We can assume that the average sizes u and v are about $n/2$ during the computations.

As we will see in the following sections, both binary extended Euclidean and Kaliski's algorithms employ the `while` loop in Algorithm 2, which determines the number of iterations needed to compute multiplicative inverse of a given integer modulo a prime number.

5 The Montgomery inversion with bEEA

In Algorithm 3, we provide a slightly modified version of the original bEEA described in [12]. The algorithm works also for odd modulus p when $\text{GCD}(a, p) = 1$. While the original algorithm directly computes $a^{-1} \bmod p$ given $a < p$, the new version computes $a^{-1}M \bmod p$, where M is an additional input of the algorithm besides a and p , and is called *the adjuster*. Also, in Step 1 of the algorithm, the initialization of s is different (compare $s \leftarrow M$ in Algorithm 3 and $s \leftarrow 1$ in the original algorithm).

The modification in the initialization step does not result in any problem as the operations applied to r and s mostly right shifts, subtractions and occasionally additions with the mod-

Algorithm 3 Montgomery Inverse with Binary Extended Euclidean Algorithm (MontInvbEEA) [12]

Input: $a \in [1, p - 1]$, p is prime and M is adjuster
Output: $r \in [1, p - 1]$ where $r = a^{-1} \cdot M \pmod{p}$

- 1: $u \leftarrow p, v \leftarrow a, r \leftarrow 0$, and $s \leftarrow M$
- 2: $k \leftarrow 0$
- 3: $\pi_0 \leftarrow (v > 0)$
- 4: **while** π_0 **do**
- 5: $\pi_1 \leftarrow (u \text{ is even})$
 $\pi_2 \leftarrow (\bar{\pi}_1 \text{ and } (v \text{ is even})),$
 $\pi_3 \leftarrow (\bar{\pi}_1 \text{ and } \bar{\pi}_2 \text{ and } (u > v)),$
 $\pi_4 \leftarrow (\bar{\pi}_1 \text{ and } \bar{\pi}_2 \text{ and } \bar{\pi}_3),$
- 6: $[\pi_1]: u \leftarrow u/2$
 if r is even **then** $r \leftarrow r/2$ **else** $r \leftarrow (r + p)/2$
- 7: $[\pi_2]: v \leftarrow v/2$
 if s is even **then** $s \leftarrow s/2$ **else** $s \leftarrow (s + p)/2$
- 8: $[\pi_3]: u \leftarrow (u - v)/2, r \leftarrow r - s$
 if r is even **then** $r \leftarrow r/2$ **else** $r \leftarrow (r + p)/2$
- 9: $[\pi_4]: v \leftarrow (v - u)/2, s \leftarrow s - r$
 if s is even **then** $s \leftarrow s/2$ **else** $s \leftarrow (s + p)/2$
- 10: $\pi_0 \leftarrow (v > 0)$
- 11: $k \leftarrow k + 1$
- 12: **end while**
- 13: **while** $r < 0$
 $r \leftarrow r + p$
- 14: **while** $r \geq p$
 $r \leftarrow r - p$
- 15: **return** r

ulus. Therefore, the integers r and s do not grow faster in the new algorithm than in the original algorithm. Consequently, we can safely state that Algorithm 3 is not less efficient than the original algorithm from the perspective of complexity. However, this modification, which is useful for the computation of Montgomery inversion, cannot be adopted in Kaliski's method as can be seen in the next section.

Using Algorithm 3, we can compute three types of Montgomery inversion algorithms. The Montgomery inversion of a number in the Montgomery domain, $a \cdot R \bmod p$ is calculated as follows

$$\begin{aligned} \text{MontInvbEEA}(a \cdot R \bmod p, p, R^2 \bmod p) \\ = a^{-1} \cdot R^{-1} \cdot R^2 = a^{-1} \cdot R \bmod p. \end{aligned}$$

As can be seen, the adjuster used in the computation is $M = R^2 \bmod p$. Similarly, we can compute the forward and backward Montgomery inverses of $a \bmod p$ and $a \cdot R \bmod p$ using the formulae

$$\begin{aligned} \text{MontInvbEEA}(a \bmod p, p, R \bmod p) \\ = a^{-1} \cdot R \bmod p, \\ \text{MontInvbEEA}(a \cdot R \bmod p, p, R \bmod p) \\ = a^{-1} \cdot R^{-1} \cdot R \bmod p = a^{-1} \bmod p, \end{aligned}$$

respectively. In both cases, $M = R \bmod p$.

Algorithm 4 Kaliski’s Almost Inversion Algorithm [13]

Input: $a \in [1, p - 1]$ and p is prime
Output: $r \in [1, p - 1]$ where $r = a^{-1} \cdot 2^k \pmod{p}$
1: $u \leftarrow p, v \leftarrow a, r \leftarrow 0$, and $s \leftarrow 1$
2: $k \leftarrow 0$
3: $\pi_0 \leftarrow (v > 0)$
4: **while** π_0 **do**
5: $\pi_1 \leftarrow (u \text{ is even})$
 $\pi_2 \leftarrow (\bar{\pi}_1 \text{ and } (v \text{ is even}))$,
 $\pi_3 \leftarrow (\bar{\pi}_1 \text{ and } \bar{\pi}_2 \text{ and } (u > v))$,
 $\pi_4 \leftarrow (\bar{\pi}_1 \text{ and } \bar{\pi}_2 \text{ and } \bar{\pi}_3)$,
6: $[\pi_1]: u \leftarrow u/2, s \leftarrow 2s$
7: $[\pi_2]: v \leftarrow v/2, r \leftarrow 2r$
8: $[\pi_3]: u \leftarrow (u - v)/2, r \leftarrow r + s, s \leftarrow 2s$
9: $[\pi_4]: v \leftarrow (v - u)/2, s \leftarrow r + s, r \leftarrow 2r$
10: $\pi_0 \leftarrow (v > 0)$
11: $k \leftarrow k + 1$
12: **end while**
13: **if** $r \geq p$ **then** $r \leftarrow r - p$
14: **return** $p - r$

6 Montgomery inversion with Kaliski’s method

Kaliski’s almost inversion algorithm [13], described in Algorithm 4, computes $a^{-1}2^k \pmod{p}$ given $a < p$. The algorithm works also for odd modulus p provided that $\text{GCD}(a, p) = 1$. As can be observed, the operations on u and v are identical to those in the binary GCD and inversion algorithms (see Algorithms 2, 3) with the only difference in the operations performed over r and s . Therefore, the expected numbers of iterations are also identical in all three algorithms.

Several analyses such as the one in [33] suggest that Kaliski’s method is more efficient than the classical EEA method based on Algorithm 1. Also as the operations over r and s are much simpler than those in binary inversion algorithm, the almost inverse algorithm (Kaliski’s method) is generally regarded as more efficient. However, the almost inversion algorithm returns $a^{-1}2^k \pmod{p}$, where $n \leq k \leq 2n$, we need a correction phase to obtain the Montgomery inversion of the input. A method for the correction phase is described in Algorithm 5, which computes the Montgomery inversion (Steps 2–10) and backward and forward Montgomery inversions (Steps 11–19).

Using Algorithms 4 and 5, we can compute all three types of Montgomery inversion algorithms. Assuming $R = 2^n \pmod{p}$ and knowing $2n \geq k \geq n$, the Montgomery inversion of integer a in the Montgomery domain, $a \cdot 2^n \pmod{p}$ is calculated as follows

$$\begin{aligned} \text{AlmInv}(a2^n \pmod{p}, p) &= a^{-1} \cdot 2^{k-n} \pmod{p} \\ \text{CorPhase}(a^{-1}2^{k-n} \pmod{p}, p, k, n) &= (a^{-1} \cdot 2^{k-n}) \cdot 2^{2n-k} \pmod{p} \\ &= a^{-1}2^n \pmod{p}. \end{aligned}$$

Algorithm 5 Correction Phase of Almost Inversion Algorithm

Input: $r = a^{-1} \cdot 2^k \pmod{p}$, k , and n
Output: $r \in [1, p - 1]$ where
 $r = a^{-1} \cdot 2^{2n} \pmod{p}$ for Montgomery inversion and
 $r = a^{-1} \cdot 2^n \pmod{p}$ for Forward and Backward Montgomery Inversions
1: $j \leftarrow 0$
2: **if** Montgomery Inversion **then**
3: $\pi_5 \leftarrow (j < 2n - k)$
4: **while** π_5 **do**
5: $r \leftarrow 2r$
6: $\pi_6 \leftarrow (r > p)$
7: $[\pi_6]: r \leftarrow r - p$
8: $j \leftarrow j + 1$
9: $\pi_5 \leftarrow (j < 2n - k)$
10: **end while**
11: **else**
12: $\pi_5 \leftarrow (j < k - n)$
13: **while** π_5 **do**
14: $\pi_6 \leftarrow (r \text{ is odd})$
15: $[\pi_6]: r \leftarrow r + p$
16: $r \leftarrow r/2$
17: $j \leftarrow j + 1$
18: $\pi_5 \leftarrow (j < k - n)$
19: **end while**
20: **end if**
21: **return** r

The forward Montgomery inversion algorithm is computed as follows

$$\begin{aligned} \text{AlmInv}(a \pmod{p}, p) &= a^{-1} \cdot 2^k \pmod{p} \\ \text{CorPhase}(a^{-1}2^k \pmod{p}, p, k, n) &= (a^{-1} \cdot 2^k) / 2^{k-n} \pmod{p} \\ &= a^{-1}2^n \pmod{p}. \end{aligned}$$

Finally, the backward Montgomery inversion algorithm is computed using exactly the same steps of Algorithm 5 as in the case of the forward Montgomery inversion algorithm.

$$\begin{aligned} \text{AlmInv}(a2^n \pmod{p}, p) &= a^{-1} \cdot 2^{k-n} \pmod{p} \\ \text{CorPhase}(a^{-1}2^{k-n} \pmod{p}, p, k, n) &= (a^{-1} \cdot 2^{k-n}) / 2^{k-n} \pmod{p} \\ &= a^{-1} \pmod{p}. \end{aligned}$$

Although the correction steps can also be implemented using several Montgomery multiplication operations as shown in [9], we prefer Algorithm 5, which consists of only addition, subtraction and shift operations (as in the cases of Algorithms 3 and 4). The fact that the solution in [9] features different types of operations, which can be more efficient than Algorithm 5 in certain platforms, renders a fair and generic comparison with the bEEA-based Montgomery inversion algorithm impossible.

An alternative method to compute Montgomery inverse using Kaliski's method is to modify Step 1 of Algorithm 4 as follows:

$$u \leftarrow p, \quad v \leftarrow a, \quad r \leftarrow 0, \quad s \leftarrow M,$$

where M is the adjuster similar to that in Algorithm 3. Although this results in a functionally correct algorithm, it will not be efficient. A close inspection on the operations on r and s in Algorithm 4 reveals that they are growing in bit size. Therefore, both r and s in the alternative solution will reach to bit sizes of about $2n$ when the algorithm terminates and performing arithmetic with double size integers is not efficient.

7 Comparing binary and Kaliski's algorithms

In this section, we compare the Montgomery inversion algorithms based on bEEA (Algorithm 3) and Kaliski's method (Algorithms 4+5) by counting the average number of iterations, shifts, and additions/subtractions. We run the algorithm 10,000 times for randomly generated parameters for bit sizes $n \in \{256, 384, 512\}$ and list the results in Table 1. We separate the results for three bit sizes using the symbol “/.” As all algorithms utilize the main loop of Algorithm 2, the average number of iterations are the same. The number of iterations in the correction phase of Kaliski's method (see Algorithm 5) is not included in these figures, while the additions and shifts due to the correction phase are all taken into account in the table (see the last three rows).

As can be observed from Table 1, the bEEA-based algorithm executes more additions than the other algorithms while its average number of shifts is always less than the others. When the Montgomery inversion is computed the total of number of shifts and additions in Algorithm 5 is slightly less than the algorithm based on Kaliski's method. One can always argue that shifts are less complex operations than additions and therefore that the Kaliski's method is more efficient. However, due to extensive hardware support for efficient addition and shift operations in many modern microprocessors, from clock count perspective this argument does not necessarily hold for multiple precision integers in all microprocessors. As there are different microprocessor architectures with different hardware support for arithmetic operations, we do not provide implementation results for any particular microprocessor. Notwithstanding, we can argue that bEEA-based algorithm and the algorithm based on Kaliski's method for computing Montgomery inverse have comparable performance.

On the other hand, for backward and forward Montgomery inverse computations, one can argue that Kaliski's method is more efficient (compare the total number of shifts and addi-

tions) provided that a multi-precision shift operation does not take significantly more time than a multi-precision addition on a given processor platform.

One issue with Algorithm 3 is that integers r and s can take negative values, while they are always positive in Algorithms 4+5. This basically means that we should use signed integer representation for r and s , which can require one additional computer word in software implementations. On the other hand, as r can become larger than the modulus p in Algorithms 4+5, we may have to use one computer word in addition to number of words used to represent p . Consequently, we can safely claim that r and s taking negative values have no adverse performance implications.

8 Parallelization of inversion algorithms

All inversion algorithms presented here utilize the main loop of Algorithm 2 which operates on u and v . The inversion algorithms, on the other hand, additionally process the secondary variables of r and s , on which types of operations differ depending on the inversion algorithm. One simple parallelization idea is to perform the operations on u and v and those on r and s in two separate computing cores concurrently. The first core, processing u and v , computes the predicates, $\pi_0\text{--}\pi_4$ in iteration i of the main loop, sends them to second core and processes u and v for that iteration. The second core, using the predicates, performs the operations on r and s concurrently. Assuming that the predicates reach the second core Δ seconds later at most, we can theoretically say (with some simplified assumptions on the idiosyncratic behavior of the computing platform otherwise) that the second core completes its computations on r and s ($\Delta - \tau_{uv} + \tau_{rs}$) seconds after the first core, where τ_{uv} and τ_{rs} are the execution times of all operations in one iteration on u and v , and r and s , respectively. We further assume that $\tau_{uv} < \Delta < k \cdot \tau_{uv}$.

The proposed parallelization method can result in a speedup (upper-bounded by 2) only on multi-core processors where data sharing among cores results in comparatively low overhead. Our experiments with processor architectures where we conclude this overhead is high, yield no speedup. We use a computer featuring six cores, with hyper-threading support running 64-bit Ubuntu Linux 12.04 operating system. Each core is an Intel Xeon CPU E1650 operating at 3.50 GHz. We utilized C++ programming language with the GMP library optimized for big number arithmetic. For parallel implementation we used the OpenMP API that allows shared-memory multiprocessing programming. As the overhead for sharing of the predicates between two cores is presumably high, we report no speedup due to parallelization for the primes of bit lengths of interest to contemporary cryptographic usage. If the proposed parallelization method

Table 1 Comparison of binary and Kaliski’s inverse algorithms for $n \in \{256, 384, 512\}$

Operation type	Algorithm 3 for all inversions	Algorithms 4+5 for Montgomery inverse	Algorithms 4+5 for backward/forward Montgomery inverse
Number of iterations	361/543/722	361/542/723	361/542/722
Ratio (k/n)	1.41/1.41/1.41	1.41/1.41/1.41	1.41/1.41/1.41
Number of additions	542/814/1085	438/656/875	415/623/829
Number of shifts	722/1086/1445	873/1310/1747	827/1243/1655
Number of (additions + shifts)	1264/1900/2529	1310/1967/2621	1243/1866/2485

turns to be useful in a multi-core processor, we speculate that binary inversion algorithm performs slightly better than Kaliski’s method as the latter requires a correction phase which is not easily parallelizable.

Finally, we do not expect that the proposed parallelization method leads to any acceleration in hardware implementations as they usually feature different functional units. However, the method can simplify the control circuit and/or the design.

9 The constant-time inversion algorithms

In this section, we propose two Montgomery inversion algorithms that take constant execution time as a protection against a class of side-channel attacks. They are basically modifications of Algorithms 3 and 4+5. The following assumptions are necessary (yet most probably not sufficient) for constant execution time of the algorithms given in the subsequent sections. Note that more assumptions may be needed depending on the hardware platform and software tool chain employed for implementation.

Assumption 1 All predicates, $\pi_0 - \pi_8$ and $\bar{\pi}_1$, are evaluated and checked once in every iteration of the main loop.

Assumption 2 When integers (u, v, r , and s) are represented as multiple precision numbers, the precision is the same for all integers.

Example 1 When p is a 256-bit integer, we represent $r = 5$ as $r = 00005$ on a 64-bit processor, where each digit is a 64-bit number. Note that we use five 64-bit words to represent 256-bit integers to capture carry out bits resulting from arithmetic operations.

Assumption 3 Arithmetic and logic operations (addition, subtraction, shift and comparison) are always executed on the full precision of integers in case they are represented as multiple precision numbers.

Example 2 Suppose p is a 256-bit integer and a 64-bit processor and we want to perform the addition of $r + s$, where

$r, s < 2^{64}$. Then we perform the following word-wise addition operations in this order

$$\begin{aligned}
 (C_0, S_0) &\leftarrow r + s \\
 (C_1, S_1) &\leftarrow C_0 + 0 \\
 (C_2, S_2) &\leftarrow 0 + 0 \\
 (C_3, S_3) &\leftarrow 0 + 0, \\
 (C_4, S_4) &\leftarrow 0 + 0,
 \end{aligned} \tag{1}$$

where C_i and S_i correspond to the carry and sum parts of the addition operation result.

Assumption 4 No register-level optimization is applied on assignment operations, such as $c \leftarrow a \text{ op } b$. The operands, a and b , are read from the memory, operation is executed, and the result, c is written back the memory. Also, the address of operands are also read from the memory.

These assumptions can only be satisfied through extensive support of assembly-level programming as software tool chains such as compilers tend to perform various optimizations that violate these assumptions. A secure implementation in assembly language is challenging, but feasible and sometimes preferred as assembly-level support in cryptographic implementations is a common practice either for speed or for security. Note that the assumptions necessitate dummy operations to be performed (e.g., Example 2), which can make the implementations of the constant-time algorithms vulnerable to fault attacks.

9.1 Constant-time inversion based on bEEA

Algorithm 6 is a version of Montgomery inversion algorithm based on bEEA (Algorithm 3), which executes in constant time independent of its inputs for a given input size ($n = \lceil \log_2 p \rceil$). More precisely, Algorithm 6 always iterates $2n$ times and every iteration features two subtractions, one addition and two shift operations. Note that the *if-then-else* statements in Steps 10-12 are dummy operations as both *if* and *else* branches perform exactly the same operations on the variable s . These steps correct the final result r whenever $r > p$ or $r < 0$, but perform dummy

Algorithm 6 Constant-Time Binary Extended Euclidean Algorithm

Input: $a \in [1, p - 1]$, p is prime, and M is adjutor
Output: $r \in [1, p - 1]$ where $r = a^{-1} \cdot M \pmod{p}$

- 1: $u \leftarrow p, v \leftarrow a, r \leftarrow 0$, and $s \leftarrow M$
- 2: $k \leftarrow 0$
- 3: $\pi_0 \leftarrow (k < 2n), \pi_1 \leftarrow (v > 0), \bar{\pi}_1 \leftarrow (v = 0)$
- 4: **while** π_0 **do**
- 5: $\pi_2 \leftarrow (u \text{ is even}), \pi_3 \leftarrow (\bar{\pi}_2 \text{ and } (v \text{ is even})),$
 $\pi_4 \leftarrow (\bar{\pi}_2 \text{ and } \bar{\pi}_3 \text{ and } (u > v)), \pi_5 \leftarrow (\bar{\pi}_2 \text{ and } \bar{\pi}_3 \text{ and } \bar{\pi}_4),$
 $\pi_6 \leftarrow (r < 0), \pi_7 \leftarrow \bar{\pi}_6 \text{ and } (r > p), \pi_8 \leftarrow \bar{\pi}_6 \text{ and } \bar{\pi}_7$
- 6: $[\pi_1 \text{ and } \pi_2]: \Delta_{uv} \leftarrow u - v, \Delta_{rs} \leftarrow r - s, \Sigma \leftarrow r + p, u \leftarrow u/2$
if r is even **then** $r \leftarrow r/2$ **else** $r \leftarrow \Sigma/2$
- 7: $[\pi_1 \text{ and } \pi_3]: \Delta_{uv} \leftarrow u - v, \Delta_{rs} \leftarrow r - s, \Sigma \leftarrow s + p, v \leftarrow v/2$
if s is even **then** $s \leftarrow s/2$ **else** $s \leftarrow \Sigma/2$
- 8: $[\pi_1 \text{ and } \pi_4]: \Delta_{uv} \leftarrow u - v, \Delta_{rs} \leftarrow r - s, \Sigma \leftarrow \Delta_{rs} + p,$
 $u \leftarrow \Delta_{uv}/2,$
if Δ_{rs} is even **then** $r \leftarrow \Delta_{rs}/2$, **else** $r \leftarrow \Sigma/2$
- 9: $[\pi_1 \text{ and } \pi_5]: \Delta_{uv} \leftarrow v - u, \Delta_{rs} \leftarrow s - r, \Sigma \leftarrow \Delta_{rs} + p,$
 $v \leftarrow \Delta_{uv}/2$
if Δ_{rs} is even **then** $s \leftarrow \Delta_{rs}/2$ **else** $s \leftarrow \Sigma/2$
- 10: $[\bar{\pi}_1 \text{ and } \pi_6]: \Delta_{uv} \leftarrow u - v, \Delta_{rs} \leftarrow s - r, r \leftarrow r + p, u \leftarrow \Delta_{uv}/2$
if s is even **then** $s \leftarrow s/2$ **else** $s \leftarrow \Sigma/2$
- 11: $[\bar{\pi}_1 \text{ and } \pi_7]: \Delta_{uv} \leftarrow u - v, r \leftarrow r - p, \Sigma \leftarrow r + s, u \leftarrow \Delta_{uv}/2$
if s is even **then** $s \leftarrow s/2$ **else** $s \leftarrow \Sigma/2$
- 12: $[\bar{\pi}_1 \text{ and } \pi_8]: \Delta_{uv} \leftarrow u - v, r \leftarrow r - s, \Sigma \leftarrow r + s, u \leftarrow \Delta_{uv}/2$
if s is even **then** $s \leftarrow s/2$ **else** $s \leftarrow \Sigma/2$
- 13: $k \leftarrow k + 1$
- 14: $\pi_0 \leftarrow (k < 2n), \pi_1 \leftarrow (v > 0), \bar{\pi}_1 \leftarrow (v = 0)$
- 15: **end while**
- 16: **return** r

operations otherwise. Therefore, the Boolean expression “ $i \neq s$ is even” must always be evaluated and protected against any compiler optimization.

9.2 Constant-time inversion using Kaliski’s method

Algorithm 7 always finishes after exactly $2n + 1$ iterations; only one addition, one subtraction, two shift operations and one assignment operation are executed in each iteration. Assuming an assignment operation is always much less complicated than a subtraction operation, we can claim that Algorithm 7 is more efficient than Algorithm 6. A similar algorithm for constant-time execution is proposed in [21], which has two disadvantages compared to Algorithm 7: (i) in every iteration, the algorithm in [21] executes one addition, one subtraction, six shifts and several predicated assignment operations, and (ii) Algorithm 7 has an integrated correction phase while the algorithm in [21] needs an extra operation for it.

Another method for computing the inverse in constant time is to use Fermat’s little theorem. For prime p , we have $a^{p-1} \equiv 1 \pmod{p}$ and $a^{p-1} = a^{p-2}a \equiv 1 \pmod{p}$; then, we obtain $a^{p-2} \equiv a^{-1} \pmod{p}$. Consequently, we can compute the Montgomery inverse of $a^{2^n} \pmod{p}$, $r \equiv a^{-1}2^{-n}2^{2^n} \equiv a^{-1}2^n \pmod{p}$. Basically, one inversion oper-

Algorithm 7 Constant-Time Algorithm Based on Kaliski’s Method

Input: $a \in [1, p - 1]$ and p is prime
Output: $r \in [1, p - 1]$ where $r = a^{-1} \cdot M \pmod{p}$

- 1: $u \leftarrow p, v \leftarrow a, r \leftarrow 0$, and $s \leftarrow 1$
- 2: $k \leftarrow 0$
- 3: $\pi_0 \leftarrow (k < 2n), \pi_1 \leftarrow (v > 0), \bar{\pi}_1 \leftarrow (v = 0)$
- 4: **while** π_0 **do**
- 5: $\pi_2 \leftarrow (u \text{ is even}), \pi_3 \leftarrow (\bar{\pi}_2 \text{ and } (v \text{ is even})),$
 $\pi_4 \leftarrow (\bar{\pi}_2 \text{ and } \bar{\pi}_3 \text{ and } (u > v)), \pi_5 \leftarrow (\bar{\pi}_2 \text{ and } \bar{\pi}_3 \text{ and } \bar{\pi}_4),$
 $\pi_6 \leftarrow (r > p), \pi_7 \leftarrow \bar{\pi}_6$
- 6: $[\pi_1 \text{ and } \pi_2]: \Delta \leftarrow r - s, \Sigma \leftarrow \Delta + s, s \leftarrow 2s, u \leftarrow u/2, r \leftarrow \Sigma$
- 7: $[\pi_1 \text{ and } \pi_3]: \Delta \leftarrow s - r, \Sigma \leftarrow \Delta + r, r \leftarrow 2r, v \leftarrow v/2, s \leftarrow \Sigma$
- 8: $[\pi_1 \text{ and } \pi_4]: \Delta \leftarrow u - v, \Sigma_{rs} \leftarrow r + s, s \leftarrow 2s, u \leftarrow \Delta/2,$
 $r \leftarrow \Sigma$
- 9: $[\pi_1 \text{ and } \pi_5]: \Delta \leftarrow v - u, \Sigma \leftarrow r + s, r \leftarrow 2r, v \leftarrow \Delta/2, s \leftarrow \Sigma$
- 10: $[\bar{\pi}_1 \text{ and } \pi_6]: \Delta \leftarrow r - p, \Sigma \leftarrow s + p, r \leftarrow 2\Delta, \Sigma \leftarrow \Sigma/2,$
 $s \leftarrow \Sigma$
- 11: $[\bar{\pi}_1 \text{ and } \pi_7]: \Delta \leftarrow r - p, \Sigma \leftarrow s + p, r \leftarrow 2r, \Sigma \leftarrow \Sigma/2,$
 $s \leftarrow \Sigma$
- 12: $k \leftarrow k + 1$
- 13: $\pi_0 \leftarrow (k < 2n), \pi_1 \leftarrow (v > 0), \bar{\pi}_1 \leftarrow (v = 0)$
- 14: **end while**
- 15: **if** $r > 0$ **then**
- 16: $\Delta \leftarrow r - p, \Sigma \leftarrow s + p, r \leftarrow 2\Delta, r \leftarrow r/2, s \leftarrow p$
- 17: **else**
- 18: $\Delta \leftarrow r - p, \Sigma \leftarrow s + p, r \leftarrow 2r, r \leftarrow r/2, s \leftarrow p$
- 19: **end if**
- 20: **return** $p - r$

ation takes approximately same time as one exponentiation operation. An exponentiation operation with a modulus of general form takes about 1.5 modular multiplications, on average. As modular multiplication is of quadratic complexity, we can claim that the proposed constant-time algorithm performs better than the method based on Fermat’s theorem for sufficiently large numbers.

Also, blinding, which is a common technique in many cryptographic applications as protection against side-channel attacks, can also be applied in inverse computation. In blinding, a random integer is used to *blind* the (sensitive/secret) inputs so that the computation is randomized, therefore not dependent on sensitive/secret values. The technique can be used as a generic countermeasure against side-channel attacks; for instance, it is used in the context of the computation of quadratic residues in a recent work [34]. In inversion, the blinding requires a random number generator and two extra multiplication operations. Suppose we want to compute the inverse of $a < p$ with respect to modulus p . Before the inversion operation, we perform the modular multiplication $\alpha = a \cdot \rho \pmod{p}$, where the blinding factor ρ is selected uniformly random in the interval $[1, p - 1]$. The inversion algorithm, which takes the randomized input α , returns $\alpha^{-1} = a^{-1} \cdot \rho^{-1} \pmod{p}$. A second modular multiplication $\alpha^{-1} \cdot \rho \pmod{p} = a^{-1} \pmod{p}$, yields the correct result. It is not straightforward to compare the blinding technique, which is effective and simple to implement, with

constant-time inversion algorithms. If performing modular multiplication operations takes a relatively small fraction of modular inversion time on a given platform, then the blinding is probably a superior approach.

10 Conclusions

We explored alternative algorithms for computing Montgomery inverse of an integer modulo a prime number. We showed that a Montgomery inversion algorithm based on binary extended Euclidean algorithm has comparable complexity to the method based on Kaliski's almost inverse algorithm. We proposed a parallelization method that can be applied to all inversion algorithms intended for software implementations on multi-core processors featuring low inter-process data sharing overhead. Finally, we introduced two efficient, constant-time algorithms for computing Montgomery inverse. The computational complexity of the proposed constant-time algorithms compares favorably with a method in the literature.

Acknowledgements We thank the anonymous reviewers for their comments and recommendations.

References

- Rivest, R.L., Shamir, A., Adleman, A.: A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **21**, 120–126 (1976)
- Diffie, W., Hellman, M.E.: New directions in cryptography. *IEEE Trans. Inf. Theory* **22**, 644–654 (1976)
- National Institute for Standards and Technology. FIPS PUB 186-4 Digital Signature Standard (DSS). doi:10.6028/NIST.FIPS.186-4 (2013)
- Koblitz, N.: Elliptic curve cryptosystems. *Math. Comput.* **48**(177), 203–209 (1987)
- Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) *Crypto 1985. Lecture Notes in Computer Science*, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
- Menezes, A.J.: *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston (1993)
- Sakai, R., Ohgishi, K., Kasahara, M.: Cryptosystems based on pairing. In: *The 2000 Symposium on Cryptography and Information Security*, Okinawa, Japan, pp. 135–148 (2000)
- Montgomery, P.L.: Modular multiplication without trial division. *Math. Comput.* **44**(170), 519–521 (1985)
- Savaş, E., Koç, Ç.K.: The Montgomery modular inverse—revisited. *IEEE Trans. Comput.* **49**(7), 763–766 (2000)
- Euclid Thirteen Books of Euclids Elements, vol. 2, Books 3–9, 2nd edn, Translated by T. L. Heath. Dover Publications (1956)
- Stein, J.: Computational problems associated with Raca algebra. *J. Comput. Phys.* **1**, 397–405 (1967)
- Knuth, D.E.: *The Art of Computer Programming*, vol. 2, 2nd edn. Addison-Wesley, Reading (1981)
- Kaliski Jr., B.S.: The Montgomery inverse and its applications. *IEEE Trans. Comput.* **44**(8), 1064–1065 (1995)
- Kobayashi, T., Morita, H.: Fast modular inversion algorithm to match any operand unit. *IEICE Trans. Fundam.* **E82-A**(5), 733–740 (1999)
- Savaş, E., Koç, Ç.K.: Architecture for unified field inversion with applications in elliptic curve cryptography. In: *Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems—ICECS 2002*, vol. 3, pp. 1155–1158. Dubrovnik, Croatia (2002)
- Lórenz, R.: New algorithm for classical modular inverse. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) *Cryptographic Hardware and Embedded Systems, LNCS*, pp. 57–70. Springer, Berlin (2002)
- Tenca, A.F., Tawalbeh, L.A.: An algorithm for unified modular division in $GF(p)$ and $GF(2^n)$ suitable for cryptographic hardware. *IEE Electron. Lett.* **40**(5), 304–306 (2004)
- Gutub, A.A.-A., Tenca, A.F., Savaş, E., Koç, Ç.K.: Scalable and unified hardware to compute Montgomery inverse in $GF(p)$ and $GF(2^n)$. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) *Cryptographic Hardware and Embedded Systems, LNCS*, pp. 485–500. Springer, Berlin (2002)
- Savaş, E., Naseer, M., Gutub, A.A.-A., Koç, Ç.K.: Efficient unified Montgomery inversion with multibit shifting. *IEE Process. Comput. Digit. Tech.* **152**(4), 489–498 (2005)
- Cohen, H., Miyaji, A., Ono, T.: Efficient elliptic curve exponentiation using mixed coordinates. In: *ASIACRYPT 1998*, pp. 51–65
- Bos, J.W.: Constant time modular inversion. *J. Cryptogr. Eng.* **4**(4), 275–281 (2014)
- Gutub, A.A.-A., Tenca, A.F., Koç, Ç.K.: Scalable VLSI architecture for $GF(p)$ Montgomery modular inverse computation. In: *IEEE Computer Society Annual Symposium on VLSI, ISVLSI'02*, pp. 46–51. Pittsburgh, Pennsylvania, USA, April 25–26 (2002)
- Gutub, A.A.-A., Tenca, A.F.: Efficient scalable hardware architecture for Montgomery inverse computation in $GF(p)$. In: *IEEE Workshop on Signal Processing Systems (SIPS'03)*, pp. 93–98. Seoul, Korea, August 27–29 (2003)
- Gutub, A.A.-A., Tenca, A.F.: Efficient scalable VLSI architecture for Montgomery inversion in $GF(p)$. *Integr. VLSI J.* **37**(2), 103–120 (2004)
- Gutub, A.A.-A., Savaş, E., Kalganova, T.: Scalable VLSI design for fast $GF(p)$ Montgomery inverse computation. In: *IEEE International Conference on Computer and Communication Engineering (ICCCE '06)*. Kuala Lumpur, Malaysia (2006)
- Gutub, A.A.-A.: High speed hardware architecture to compute galois fields $GF(p)$ montgomery inversion with scalability features. *IET Comput. Digit. Tech.* **1**(4), 389–396 (2007)
- Zi-bin, D., Fan, Q., Xiao-hui, Y.: Scalable hardware architecture for montgomery inversion computation in dual-field. In: *2009 WASE International Conference on Information Engineering*, pp. 206–209. Taiyuan, Chanxi (2009)
- Chen, C., Qin, Z.: Efficient algorithm and systolic architecture for modular division. *Int. J. Electron.* **98**(6), 813–823 (2011)
- Murat, E., Kardaş, S., Savaş, E.: Scalable and efficient FPGA implementation of Montgomery inversion. In: *Proceedings of the 2011 Workshop on Lightweight Security and Privacy: Devices, Protocols, and Applications, LIGHTSEC'11*, pp. 61–68 (2011)
- Liu, Z., Wenger, E., Großschädl, J.: MoTE-ECC: energy-scalable elliptic curve cryptography for wireless sensor networks. In: *ACNS 2014*, pp. 361–379
- 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. *IACR Cryptol. ePrint Arch.* **2016**, 365 (2016)
- Aranha, D.F., Fuentes-Castañeda, L., Knapp, E., Menezes, A., Rodríguez-Henríquez, F.: Implementing Pairings at the 192-Bit Security Level, pp. 177–195. Pairing (2012)
- De Win, E., Mister, S., Preneel, B., Wiener, M.: On the performance of signature schemes based on elliptic curves. In: Buhler,

- J.P. (ed) *Algorithmic Number Theory: Third International Symposium, ANTS-III*, pp. 252–266. Portland, Oregon, USA, June 21–25, Springer, Berlin (1998)
34. Fouque, P.-A., Tibouchi, M.: Indifferentiable hashing to Barreto–Naehrig curves. In: *LATINCRYPT 2012*, pp. 1–17