This article is the final version submitted by the author to Springer-Verlag in August 2014.
The version published by Springer-Verlag in the Journal of Cryptographic Engineering is available at http://dx.doi.org/10.1007/s13389-014-0084-8.

Constant Time Modular Inversion

Joppe W. Bos

Abstract Simple power analysis is a common technique to attack software implementations, especially in the realm of public-key cryptography. An effective countermeasure to protect an implementation is to ensure constant (worst-case) runtime. In this paper we show how to modify an algorithm by Kaliski to compute the Montgomery inverse such that it can compute both the classical and Montgomery modular inverse in constant time. We demonstrate the effectiveness by comparing it to the approach based on Fermat's little theorem as used in the current simple power analysis resistant implementations in cryptography. Our implementation on the popular 32-bit ARM platform highlights the practical benefits of this algorithm.

1 Introduction

In recent years it has become essential to guard (software) implementations against physical attacks. This research area, related to side-channel resistance of cryptographic primitives, started with the introduction of *simple power analysis* techniques [14] which measure physical information of the device on which the implementation is running. This enables an attacker to obtain the secret material from a (possibly small) number of observations when some of the characteristics of the implementation depend on the input. An important counter-measure against such attacks is to ensure that the implementation has *constant run-time*: i.e. the execution time of the implementation does not depend on the input. In practice this is usually realized by eliminating all code that contains data-dependent branches.

Part of this work was performed while working for Microsoft Research

NXP Semiconductors, Leuven, Belgium

A popular approach to realize public-key cryptography is based on the algebraic structure of elliptic curves over finite fields. This is known as elliptic curve cryptography [13,17] which continues to enjoy increasing popularity since its invention in the mid 1980s. The attractiveness of small key-sizes [15] has placed this public-key primitive as the preferred alternative to RSA [22]. Standardized public-key algorithms (e.g. [28]) based on elliptic curves, defined over prime fields $\mathbf{F}_p$ with $p > 3$, involve computing a scalar multiple of a point on an elliptic curve. Such a scalar multiplication can be computed by calculating a sequence of affine group operations, each of which can be implemented by a number of multiplications and an inversion modulo p. In order to speed up this computation, projective coordinates are used in practice. This has the advantage of trading the computationally expensive modular inversion operation in each group operation for modular multiplications. Typically, at the end of the scalar multiplication, conversion from projective to affine coordinates is realized at the cost of a single modular inversion.

Another application of elliptic curves is to instantiate a powerful cryptographic primitive: the bilinear pairing [10,23,3]. Just as in the setting of elliptic curve cryptography, pairing-based protocols typically compute (at least) one modular inversion per pairing computation.

In [19] it is shown that the projective representation of an elliptic curve point can reveal information about the secret. Hence, the modular inversion computation, required to convert from projective to affine coordinates, depends on data which can reveal the secret and therefore can leak information. This means that the time to compute the modular inversion should be independent of the number to be inverted (but may depend on the prime modulus).

Recent constant time performance implementations of curve based cryptography realize this inversion modulo a prime p in constant time using Fermat's little theorem. When an inversion of an integer a is required one can compute the exponentiation $a^{p-2} \equiv a^{-1} \bmod p$. This is done in constant time using addition chains [25] with the help of the square-and-multiply algorithm. For instance, inversions modulo the prime $2^{255} - 19$ take seven percent of the entire run-time of the scalar multiplication and can be computed using 254 modular squarings and 11 modular multiplications [2]. The same approach, and similar performance numbers, are reported in recent curve based approaches which protect themselves against simple power analysis and run in constant time [20, 16, 4, 8, 5].

In this paper we propose a different approach to compute modular inversions in constant time. We modify the (non-constant time) approach from [11], which is in part based on [9], which computes the Montgomery inverse based on the extended binary greatest common divisor algorithm (see [26] and [12, Ex. 39, Sec. 4.5.2]). We show how to use this algorithm to compute both regular modular inversions as well as computing inversions when using Montgomery arithmetic [18]. We show that for moduli which do not have a "special" shape this approach is faster than computing the modular inverse using Fermat's little theorem. Our target platform is the ARM: a popular 32-bit platform which can be found on many modern mobile and embedded devices.

This paper is organized as follows. In Section 2 we recall the necessary background on Montgomery arithmetic. Section 3 describes the "almost" Montgomery inversion method by Kaliski. We outline the modified algorithm which runs in constant time in Section 4. Section 5 discusses implementations of constant time modular inversions on the ARM platform and Section 6 concludes the paper.

2 Montgomery Arithmetic

The Montgomery modular multiplication method [18] transforms each of the operands into a Montgomery residue and carries out the computation by replacing the conventional modular multiplications by Montgomery multiplications. Due to the overhead of changing representations, Montgomery arithmetic performs best when used to replace a sequence of modular multiplications, since the overhead is amortized. This is suitable to speed up, for example, modular exponentiations which can be decomposed as a sequence of several modular multiplications.

The idea behind Montgomery multiplication is to replace the expensive division operations required when computing the modular reduction by the much cheaper shift operations (division by powers of two) on computer architectures. Let w denote the word size in bits. We write integers in a radix-2^w system, where typical values of w are $w = 32$ or $w = 64$ to match the word-size of modern computer platforms. Let m be an n-word odd modulus such that $2^{w(n-1)} \leq m < 2^{wn}$. The Montgomery radix 2^{wn} is a constant such that it is co-prime to m (which is why we require m to be odd). The Montgomery residue of an integer $a \in \mathbf{Z}/m\mathbf{Z}$ is defined as $\widetilde{a} = a \cdot 2^{wn} \bmod m$. The Montgomery multiplication of two residues is defined as $M(\widetilde{a}, \widetilde{b}) = \widetilde{a} \cdot \widetilde{b} \cdot 2^{-wn} \bmod m$. Residues may be added and subtracted using regular modular algorithms since

$$\widetilde{a} \pm \widetilde{b} \equiv (a \cdot 2^{wn}) \pm (b \cdot 2^{wn}) \equiv (a \pm b) \cdot 2^{wn} \pmod{m}.$$

The Montgomery product $\widetilde{c} = M(\widetilde{a}, \widetilde{b})$ can be computed in two steps. First calculate the integer product $d = \widetilde{a}\widetilde{b}$. Next perform the *Montgomery reduction* with the help of a w-bit precomputed value $\mu = -m^{-1} \bmod 2^w$: compute $\frac{d}{2^{wn}} \bmod m$ by replacing n times in succession d by

$$\frac{d + ((d\mu) \bmod 2^w)m}{2^w},$$

then $\widetilde{c} = d - m$ if $d \geq m$ and $\widetilde{c} = d$ otherwise. If $0 \leq \widetilde{a}, \widetilde{b} < m$, then the same bounds hold for $\widetilde{c}$. A common technique to avoid this conditional subtraction in the Montgomery multiplication algorithm, which is computed to ensure the result is properly reduced such that it can be used as input to the algorithm again, is to use a redundant representation. When the modulus m is chosen such that $4m < 2^{wn}$, then the inputs and output are represented as elements of $\mathbf{Z}/2m\mathbf{Z}$ instead of $\mathbf{Z}/m\mathbf{Z}$. It is easily shown that throughout the series of modular multiplications, outputs from multiplications can be reused as inputs, and these values remain bounded without the need to compute this conditional subtraction [29].

Computing the Montgomery inverse $a^{-1} \cdot 2^{wn} \bmod m$ of a Montgomery residue $\widetilde{a} = a \cdot 2^{wn} \bmod m$ can be done by computing a modular inversion and a modular multiplication. First compute $\widetilde{a}^{-1} \equiv a^{-1} \cdot 2^{-wn} \pmod{m}$ and subsequently correct this value by multiplying by 2^{2wn} or 2^{3wn} modulo m using regular or Montgomery multiplication, respectively. The computation of the Montgomery inverse is studied in [11, 24], while a modified version of this algorithm is shown to be suitable for computation on platforms which support the 4-way single instruction, multiple data paradigm [6].

Algorithm 1 Compute the "almost" modular inverse $b^{-1} \cdot 2^k \bmod a$ [11].

Input: $a, b \in \mathbf{Z}_{>0}$ with $\gcd(a, b) = 1$ and $0 \leq b < a$.
Output: $\begin{cases} (b^{-1} \cdot 2^k \bmod a, k) \text{ where} \\ \lceil \log_2(a) \rceil \leq k \leq 2\lceil \log_2(a) \rceil. \end{cases}$

$u \leftarrow a,\ v \leftarrow b,\ r \leftarrow 0,\ s \leftarrow 1,\ k \leftarrow 0$
while $v \neq 1$ **do**
  if $u \equiv 0 \pmod 2$ **then**
    $u \leftarrow u/2,\ s \leftarrow 2 \cdot s$
  else if $v \equiv 0 \pmod 2$ **then**
    $v \leftarrow v/2,\ r \leftarrow 2 \cdot r$
  else if $u > v$ **then**
    $u \leftarrow (u - v)/2,\ r \leftarrow r + s,\ s \leftarrow 2 \cdot s$
  else
    $v \leftarrow (v - u)/2,\ s \leftarrow r + s,\ r \leftarrow 2 \cdot r$
  end if
  $k \leftarrow k + 1$
end while
return (s, k)

3 The Almost Montgomery Inverse

An "almost" modular inversion algorithm is introduced by Kaliski in [11] in the setting of computing the Montgomery inverse. This approach is outlined in Algorithm 1 to compute $b^{-1} \cdot 2^k \bmod a$ for co-prime positive integers a and b where $\lceil \log_2(a) \rceil \leq k \leq 2\lceil \log_2(a) \rceil$. The algorithm has the following invariants [11]: if $a > b > 0$, then

$$1 \leq s \leq a, \quad 1 \leq u \leq a, \quad 0 \leq v \leq b, \quad 0 \leq r < 2a.$$

This algorithms is called the "almost" modular inversion algorithm because it outputs the inverse multiplied with this power of two. The factor $2^k \bmod a$ can be removed by table look-up of the value $2^{-k} \bmod a$ and performing a modular multiplication (or precomputing the appropriate value when using Montgomery multiplication as outlined in Section 2). This can also be computed by applying

$$s_i = \begin{cases} \dfrac{s_{i-1}}{2}, & \text{if } s_{i-1} \text{ is even,} \\ \dfrac{s_{i-1} + a}{2}, & \text{if } s_{i-1} \text{ is odd,} \end{cases}$$

k times (starting at $i = 1$) and initializing $s_0 = b^{-1} \cdot 2^k \bmod a$ (the return value of Algorithm 1). This ensures that all values in the numerator are even (since a is odd).

Analyzing the worst-case number of iterations of the while-loop in Algorithm 1 is not difficult. Every iteration removes at least a factor of two, from either u or v, and the maximum number of iterations is therefore $2\lceil \log_2(a) \rceil$ (see [11]). Similarly, the minimum number of iterations is $\lceil \log_2(a) \rceil$ (the bit-length of the modulus a). This explains the bounds on the exponent k when the algorithm terminates and the size of the lookup table for the removal of the power of two at the end of the algorithm.

Algorithm 2 The constant time version of the "almost" modular inverse $b^{-1} \cdot 2^k \bmod a$ algorithm. Comments showing which branch from Algorithm 1 are being computed are displayed after a '#'.

Input: $a, b \in \mathbf{Z}_{>0}$ with $\gcd(a, b) = 1$ and $0 \leq b < a$.
Output: $\begin{cases} (b^{-1} \cdot 2^k \bmod a, k) \text{ where} \\ \lceil \log_2(a) \rceil \leq k \leq 2\lceil \log_2(a) \rceil. \end{cases}$

$u \leftarrow a,\ r \leftarrow 0,\ v \leftarrow b,\ s \leftarrow 1,\ \mathbf{k} \leftarrow 0$
for i $= 1$ to $2\lceil \log_2(a) \rceil$ **do**
  $\mathbf{uv}_< \leftarrow \text{sub}(u', u, v)$
  $\mathbf{uv}_= \leftarrow \text{equal}(u', 0)$
  $\mathbf{d} \leftarrow 0 - \mathbf{uv}_=$ $\quad \# \ \mathbf{d} = \begin{cases} 0 & \text{if } u \neq v \\ 2^w - 1 & \text{if } u = v \end{cases}$

  $\left.\begin{array}{l} \text{lshift}_1(\tilde{s}, s) \\ \text{add}(rs, r, s) \\ \text{rshift}_1(\tilde{u}, u) \\ \mathbf{m_1} \leftarrow \mathbf{d} \vee (0 - (\mathbf{u_0} \wedge 1)) \\ \mathbf{m_2} \leftarrow \text{bitflip}(\mathbf{m_1}) \\ \text{select}(u, \tilde{u}, \mathbf{m_2}, u, \mathbf{m_1}) \\ \text{select}(s, \tilde{s}, \mathbf{m_2}, s, \mathbf{m_1}) \end{array}\right\}$ $\begin{array}{l} \# \ \textbf{if } u \equiv 0 \pmod 2 \\ \# \quad u \leftarrow u/2 \\ \# \quad s \leftarrow 2 \cdot s \end{array}$

  $\left.\begin{array}{l} \text{lshift}_1(\tilde{r}, r) \\ \mathbf{S} \leftarrow (\mathbf{d} \vee \text{bitflip}(\mathbf{m_1})) \\ \mathbf{m_3} \leftarrow S \vee 0 - (\mathbf{v_0} \wedge 1) \\ \mathbf{m_4} \leftarrow \text{bitflip}(\mathbf{m_3}) \\ \text{rshift}_1(\tilde{v}, v) \\ \text{select}(v, \tilde{v}, \mathbf{m_4}, v, \mathbf{m_3}) \\ \text{select}(r, \tilde{r}, \mathbf{m_4}, r, \mathbf{m_3}) \end{array}\right\}$ $\begin{array}{l} \# \ \textbf{else if } v \equiv 0 \pmod 2 \\ \# \quad v \leftarrow v/2 \\ \# \quad r \leftarrow 2 \cdot r \end{array}$

  $\left.\begin{array}{l} \mathbf{S} \leftarrow \mathbf{S} \vee \text{bitflip}(\mathbf{m_3}) \\ \mathbf{m_5} \leftarrow \mathbf{S} \vee (0 - \mathbf{uv}_<) \\ \mathbf{m_6} \leftarrow \text{bitflip}(\mathbf{m_5}) \\ \text{rshift}_1(u', u') \\ \text{select}(u, u', \mathbf{m_6}, u, \mathbf{m_5}) \\ \text{select}(r, rs, \mathbf{m_6}, r, \mathbf{m_5}) \\ \text{select}(s, \tilde{s}, \mathbf{m_6}, s, \mathbf{m_5}) \end{array}\right\}$ $\begin{array}{l} \# \ \textbf{else if } u > v \\ \# \quad u \leftarrow (u - v)/2 \\ \# \quad r \leftarrow r + s \\ \# \quad s \leftarrow 2 \cdot s \end{array}$

  $\left.\begin{array}{l} \mathbf{S} \leftarrow \mathbf{S} \vee \text{bitflip}(\mathbf{m_5}) \\ \mathbf{m_7} \leftarrow \text{bitflip}(\mathbf{S}) \\ \text{sub}(\tilde{v}, v, u) \\ \text{rshift}_1(\tilde{v}, \tilde{v}) \\ \text{select}(v, \tilde{v}, \mathbf{m_7}, v, \mathbf{S}) \\ \text{select}(s, rs, \mathbf{m_7}, s, \mathbf{S}) \\ \text{select}(r, \tilde{r}, \mathbf{m_7}, r, \mathbf{S}) \end{array}\right\}$ $\begin{array}{l} \# \ \textbf{else} \\ \# \quad v \leftarrow (v - u)/2 \\ \# \quad s \leftarrow r + s \\ \# \quad r \leftarrow 2 \cdot r \end{array}$

  # Update or keep the current exponent value k
  $\mathbf{k} \leftarrow ((\mathbf{k} \wedge \mathbf{d}) \vee ((\mathbf{k} + \mathbf{1}) \wedge \text{bitflip}(\mathbf{d})))$
end for
return $(s, \mathbf{k})$

4 Constant Time Modular Inversion

The execution time of Algorithm 1 depends on both the value of a and b. The number of iterations of the while-loop as well as the powers of two that need to be removed at the end of the algorithm differ significantly

for different inputs. In order to turn Algorithm 1 into a constant time algorithm, for a given wn-bit modulus m, we have to ensure

1. that a single iteration is always computed in the same amount of time. This means computing *all four* different branches from Algorithm 1 and selecting the correct values in constant time. This ensures that the run-time of a single iteration is independent of the branch taken but means the computation time is increased to (at most) the time to compute the sum of all four branches.
2. that the algorithm always computes the same number of (constant time) iterations. This implies that the computation of the worst-case number of $2wn$ iterations is always required. This can be realized by detecting when Algorithm 1 would have terminated (when we reach $v = 1$). Depending on this condition we create a bit-mask and select either the input value to this iteration (when $v = 1$) or the values computed on by the constant time iteration (when $v \neq 1$).

Let us first set the notation we use throughout this section. We distinguish between w-bit integers (denoted in **bold**) which are typically used in masking operations and full wn-bit integers (denoted in regular font). We use the following w-bit masks: $\mathbf{d}$, $\mathbf{S}$, $\mathbf{m_1}$, $\mathbf{m_2}$, $\mathbf{m_3}$, $\mathbf{m_4}$, $\mathbf{m_5}$, $\mathbf{m_6}$, $\mathbf{m_7}$, which contain either all-zero bits (the integer value 0) or all-one bits (the integer value $2^w - 1$). We write the wn-bit integers in a radix-2^w system: $u = \sum_{i=0}^{n-1} \mathbf{u}_i 2^{wi}$. In the algorithm we include some optimizations which can be applied on the lower (bit) level: e.g. $u \bmod 2 = \mathbf{u_0} \wedge 1$. We use the following (constant time) functions where the (partial) output of the functions is always written as the first parameter.

- The selection function is denoted by $\text{select}(c, a, m_1, b, m_2)$ and computed as

$$c \leftarrow \begin{cases} b & \text{if } m_1 = 0 \text{ and } m_2 = 2^w - 1, \\ a & \text{if } m_1 = 2^w - 1 \text{ and } m_2 = 0, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

  That is, depending on the value of two bit-masks m_1 and m_2 the value a or b is assigned to c. In practice this functionality can be implemented in constant time in different ways. For instance, $\text{select}(c, a, m_1, b, m_2)$ can be computed as

$$c \leftarrow ((a \wedge m_1) \vee (b \wedge m_2))$$

  using two `and` instructions and a single `or` instruction per w-bit limb.

- The subtraction function is denoted by $\text{sub}(z, x, y)$ (and addition as $\text{add}(z, x, y)$) and computes $z \leftarrow x - y$ and return the borrow: i.e. one if $x < y$ and zero otherwise (while addition computes $z \leftarrow x + y$).
- The equality function is denoted by $\text{equal}(x, y)$ and returns one if $x = y$ and zero otherwise. This can be computed in constant time for wn-bit integers x and y using

  $r \leftarrow (x_0 = y_0)$
  for $i = 1$ to $n - 1$ **do**
  $\quad r \leftarrow r \wedge (x_i = y_i)$
  end for
  return r

- The shift-by-one functions are denoted by $\text{lshift}_1(z, x)$ and $\text{rshift}_1(z, x)$ for shifting x one bit position to the left or right respectively (shifting in a zero) and storing the result in z.
- The bitflip function is denoted by $\text{bitflip}(x)$ simply flips all of the bits in x and returns the result.

Algorithm 2 outlines the constant time version of Algorithm 1. All functions operate on integer values of fixed length (either w- or wn-bit), this is to ensure constant running time. In a non-constant time implementation one can use the fact that r and s start small (use only a single computer word) and can be computed on with more efficient arithmetic routines. Note that all of the arithmetic operations performed on the w-bit masks work with a single computer word and are independent of the size of the modulus. We indicated the computation of the different branches from Algorithm 1 for the ease of readability in Algorithm 2 (text after the '#' character should be regarded as comment).

The bit-mask d in Algorithm 2 indicates if the non-constant time algorithm (Algorithm 1) would have terminated since

$$\mathbf{d} = \begin{cases} 0 & \text{if } u \neq v \\ 2^w - 1 & \text{if } u = v \end{cases}$$

Hence, d contains w ones if the non-constant time algorithm would have terminated or w zeros if not. This bit-mask d is used as one of the main control parameters for the values to be selected. If d is set to all ones, then the values from the beginning of the iteration are selected as the desired result at the end of the iteration, so no modifications are made. However, all of the computations are performed to ensure the constant run-time. The four pairs of bit-masks $\{m_1, m_2\}$, $\{m_3, m_4\}$, $\{m_5, m_6\}$, and $\{S, m_7\}$ are used to differentiate between the four different branches and select the correct result.

Note that some computations can be saved. For instance, the value $r + s$ (which is either stored in r or s)

has to be computed once only. The same holds for the shifting operations, multiplying or dividing by two, but there is a trade-off between reducing these arithmetic costs and the number of calls to the select function that need to be performed. These optimizations have been included in Algorithm 2.

5 Implementation Results

We have implemented the constant time modular inversion algorithm as described in Algorithm 2 for the popular 32-bit ARM architecture which can be found in many mobile and embedded devices. In this section we first determine the speed of constant time implementations based on Fermat's little theorem and next we compare this to the performance of the constant time almost Montgomery inverse algorithm.

5.1 Fermat's Little Theorem

The current state-of-the-art proposals and software implementations of methods and protocols which are based on (elliptic) curve cryptography all use primes of a special form to enhance the practical performance of the modular arithmetic [2,16,4,8,5]. In the context of computing bilinear pairings the latest cryptographic implementations do not use special primes but have other design choices which make these efficient for software implementations. Also in this setting it is not uncommon to implement the various arithmetic in constant time (cf. [20]). The standard way of computing the modular inversion in constant time, and used in all the aforementioned implementation projects, is based on Fermat's little theorem. To compute the modular inverse $x^{-1} \bmod m$, for a positive prime number m, one can use Fermat's little theorem which states that $x^m - x$ is an integer multiple of m. Hence, one can use the identity

$$x^{-1} \equiv x^{m-2} \bmod m.$$

Using the basic square-and-multiply algorithm for modular exponentiation, this requires, for a random wn-bit prime m, roughly wn modular squarings and $wn/2$ modular multiplications.

In the setting of computing an inverse modulo a special prime these numbers can be significantly improved as illustrated in the following example.

Example 1 Computing the constant time inversion modulo the prime $2^{255} - 19$ can be realized using only 254 modular squarings and 11 modular multiplications as proposed by Daniel J. Bernstein in [2]. One can compute

$$a = z^{-1} \equiv z^{2^{255}-21} \bmod 2^{255} - 19$$

using the following steps (where the value after the # denotes the value of the exponent).

$$\begin{array}{lcll}
z_2 & \leftarrow & z^2 & \#\ 2 \\
z_9 & \leftarrow & z_2^{2^2} \cdot z & \#\ 9 \\
z_{11} & \leftarrow & z_9 \cdot z_2 & \#\ 11 \\
t_1 & \leftarrow & z_{11}^2 \cdot z_9 & \#\ 2^5 - 2^0 \\
t_2 & \leftarrow & t_1^{2^5} \cdot t_1 & \#\ 2^{10} - 2^0 \\
t_3 & \leftarrow & t_2^{2^{10}} \cdot t_2 & \#\ 2^{20} - 2^0 \\
t_4 & \leftarrow & (t_3^{2^{20}} \cdot t_3)^{2^{10}} \cdot t_2 & \#\ 2^{50} - 2^0 \\
t_5 & \leftarrow & t_4^{2^{50}} \cdot t_4 & \#\ 2^{100} - 2^0 \\
a & \leftarrow & ((t_5^{2^{100}} \cdot t_5)^{2^{50}} \cdot t_4)^{2^5} \cdot z_{11} & \#\ 2^{255} - 21
\end{array}$$

5.2 Performance Comparison

We have implemented the algorithm outlined in Algorithm 2 for the 32-bit ARM architecture. Given an integer b and an wn-bit prime modulus a, the algorithm returns $b^{-1} \cdot 2^k \bmod a$ after exactly $2wn$ iterations. These powers of two can be removed one at a time as outlined in Section 3. This can be time-consuming when it needs to be computed in constant time. We choose to remove this value using a *single* modular multiplication from a precomputed table of wn elements of wn-bit each. The difficulty is not to perform this modular multiplication in constant time but to extract the correct value from this table such that we do not leak information. Attacks that use this type of information are known as cache-attacks [21,27], which are able to deduce information from the memory access pattern. In order to guard against such attacks we access *every* element in the table and select (mask) the correct value. This results only in a insignificant overhead compared to the overall modular inversion computation.

Our benchmark platform is the BeagleBoard-xM [1], a low-power open-source hardware single-board computer, which contains the TI DM3730 system on chip (SoC) equipped with a 1.0 GHz Cortex-A8 ARM core. We implemented the various routines using the C programming language with the help of intrinsics.

In order to investigate the performance difference between the classical and constant-time version of the almost Montgomery inversion routine we benchmarked both routines for various modulus sizes. These results are summarized in Table 1. Note that we benchmarked moduli of $wn - 2$ bits to accommodate the usage of the subtraction less Montgomery multiplication (see Section 2) which runs inherently in constant-time and is more efficient in practice (this are the typical bit lengths used in bilinear pairing record setting software implementations).

The average run time of the binary version of the (extended) greatest common divisor algorithm is stud-

Table 1 The number of 10^3 cycles required to compute the modular inverse using the regular (non constant time) method (Algorithm 1) and the constant time method (Algorithm 2) using a b-bit modulus. The estimated and real number of iterations for Algorithm 1 are given as well. The number of iterations for the constant time algorithm is always $2b$.

bitsize b	# 10^3 cycles constant-time	# 10^3 cycles regular	# iterations practice	# iterations theory
254	486	57	358.1	358.6
446	1472	182	627.9	629.7
638	3028	389	898.7	900.8

ied in [12] and based on the analysis from [7]. This analysis shows that Algorithm 1 has an estimated average number of iterations of $1.41194wn$. These estimates and the real values, obtained after averaging 10 runs of 1000 trials, are included in Table 1 as well.

Based on the increased number of iterations (from $1.41194wn$ to $2wn$) and the increased cost of the iteration (we have to compute all four branches every time) one might expect a slow-down of the constant-time implementation of a factor 5.7 compared to the classical algorithm. As Table 1 shows this value is consistently higher and around a factor eight. This can be explained due to the fact that the selection of the correct values, using bitmasks, is not for free and incurs an additional performance penalty for each of the four branches.

Table 2 summarizes the performance cost on our benchmark platform for generic b-bit prime moduli p. We assume that computing the exponentiation with $p-2$ is computed using the straight-forward multiply-and-square algorithm which requires approximately b modular squarings and $b/2$ modular multiplications. Table 2 highlights the performance increase of Algorithm 2 over an approach based on Fermat's little theorem. The performance gain is more noticeable for larger moduli.

Note, however, that we do not expect this approach to be (significantly) faster compared to primes of a special shape. Let's consider the prime $2^{255} - 19$ again as an example, the 254 modular squarings and 11 modular multiplications can be computed in 406 thousand cycles using Montgomery arithmetic. This already outperforms Algorithm 2 but we expect that an implementation which takes advantage of the special shape of this prime can outperform Montgomery arithmetic by up to a factor two. It remains an interesting case study to compare Algorithm 2 to methods based on Fermat's little theorem in the setting of much larger primes of a special shape (primes which are over 512 bits).

Table 2 Performance numbers in 10^3 cycles for inversion modulo a generic (not of a special form) b-bit prime modulus. For the methods labeled "Fermat" we assume that b modular squarings and $b/2$ modular multiplications are used.

b	Fermat	Algorithm 2
254	584	486
446	2916	1472
638	8083	3028

6 Conclusions and Future Work

We have shown how to modify an algorithm by Kaliski [11] to compute the classical modular inversion as well as the Montgomery inversion in constant time. This has applications in public-key cryptography, where constant running time is an important counter-measure against simple power analysis attacks. We have shown that on the popular ARM architecture this approach outperforms the current approaches which are based on Fermat's little theorem when "generic" prime moduli are used. In the setting where primes of special shape can be used the modular inversion approach based on Fermat's little theorem might be more efficient.

We hope that the results presented in this paper inspire other people to investigate different variants of the binary GCD algorithm to see if they can be turned into more efficient versions than the one presented here.

References

1. Beagle Board. BeagleBoard-xM System Reference Manual. `http://beagleboard.org/static/BBxMSRM_latest.pdf`, 2013.
2. Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, Heidelberg, 2006.
3. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001.
4. Joppe W. Bos, Craig Costello, Huseyin Hisil, and Kristin Lauter. Fast cryptography in genus 2. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 194–210. Springer Berlin Heidelberg, 2013.
5. Joppe W. Bos, Craig Costello, Huseyin Hisil, and Kristin Lauter. High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture*

Notes in Computer Science, pages 331–348. Springer, Heidelberg, 2013.
6. Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *International Journal of Applied Cryptography*, 2(3):212–228, 2012.
7. Richard P. Brent. Analysis of the binary Euclidean algorithm. In J. F. Traub, editor, *New Directions and Recent Results in Algorithms and Complexity*, pages 321–355. Academic Press, 1976.
8. Armando Faz-Hernandez, Patrick Longa, and Ana H. Sanchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. Cryptology ePrint Archive, Report 2013/158, 2013. `http://eprint.iacr.org/`.
9. Alain Guyot. OCAPI: architecture of a VLSI coprocessor for the GCD and the extended GCD of large numbers. In *IEEE Symposium on Computer Arithmetic*, pages 226–231. IEEE, 1991.
10. Antoine Joux. A one round protocol for tripartite Diffie-Hellman. *Journal of Cryptology*, 17(4):263–276, 2004.
11. Burton S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.
12. Donald E. Knuth. *Seminumerical Algorithms*. The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, USA, 3rd edition, 1997.
13. Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.
14. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Crypto 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, Heidelberg, 1996.
15. Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.
16. Patrick Longa and Francesco Sica. Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 718–739. Springer, 2012.
17. Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Crypto 1985*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, Heidelberg, 1986.
18. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
19. David Naccache, Nigel P. Smart, and Jacques Stern. Projective coordinates leak. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 257–267. Springer, 2004.
20. Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In Michel Abdalla and Paulo S.L.M. Barreto, editors, *Progress in Cryptology – LATINCRYPT 2010*, volume 6212 of *Lecture Notes in Computer Science*, pages 109–123. Springer-Verlag Berlin Heidelberg, 2010.
21. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.
22. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
23. R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems based on pairing. In *The 2000 Symposium on Cryptography and Information Security, Okinawa, Japan*, pages 135–148, 2000.
24. Erkay Savas and Ç. K. Koç. The montgomery modular inverse-revisited. *IEEE Transactions on Computers*, 49(7):763–766, 2000.
25. A. Scholz. Aufgabe 253. *Jahresbericht der deutschen Mathematiker-Vereingung*, 47:41–42, 1937.
26. Josef Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397–405, 1967.
27. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.
28. U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-3, 2009. `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`.
29. Colin D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21):1831–1832, 1999.