Montgomery Multiplication Using Vector Instructions

Joppe W. Bos, Peter L. Montgomery, Daniel Shumow, and Gregory M. Zaverucha

Microsoft Research
August 20, 2013

Abstract. In this paper we present a parallel approach to compute *interleaved* Montgomery multiplication. This approach is particularly suitable to be computed on 2-way single instruction, multiple data platforms as can be found on most modern computer architectures in the form of vector instruction set extensions. We have implemented this approach for tablet devices which run the x86 architecture (Intel Atom Z2760) using SSE2 instructions as well as devices which run on the ARM platform (Qualcomm MSM8960, NVIDIA Tegra 3 and 4) using NEON instructions. When instantiating modular exponentiation with this parallel version of Montgomery multiplication we observed a performance increase of more than a factor of 1.5 compared to the sequential implementation in OpenSSL for the classical arithmetic logic unit on the Atom platform for 2048-bit moduli.

Key words: Montgomery multiplication, SIMD, software implementation, vector instructions

1 Introduction

Modular multiplication of large integers is a computational building block used to implement public-key cryptography. For schemes like RSA [34], ElGamal [11] or DSA [36], the most common size of the modulus for parameters in use is large; 1024 bits long [28, 20]. The typical modulus size will increase to 2048 and 3072 bits over the coming years, in order to comply with the current 112- and 128-bit security standard (cf. [31]). When computing multiple modular multiplications, Montgomery multiplication [30] provides a speed up to this core arithmetic operation. As RSA-based schemes are arguably the most frequently computed asymmetric primitives today, improvements to Montgomery multiplication are of immediate practical importance.

Many modern computer architectures provide vector instruction set extensions in order to perform single instruction, multiple data (SIMD) operations. Example platforms include the popular x86 architecture as well as the ARM platform that can be found in almost all modern smartphones and tablets. The research community has studied ways to reduce the latency of Montgomery multiplication by parallelizing this computation. These approaches vary from using the SIMD paradigm [10, 23, 8, 18] to the single instruction, multiple threads paradigm using a residue number system [14, 29] as described in [4, 19] (see Section 2.3 for a more detailed overview).

In this paper we present an approach to split the Montgomery multiplication into two parts which can be computed in parallel. We flip the sign of the precomputed Montgomery constant and accumulate the result in two separate intermediate values that are computed concurrently. This avoids using a redundant representation, for example suggested in the recent SIMD approach for Intel architectures [18], since the intermediate values do not overflow to an additional word. Moreover, our approach is suitable for implementation using vector instruction set extensions which support 2-way SIMD operations, i.e., a single instruction that is applied to two data segments simultaneously. We implemented the sequential Montgomery multiplication algorithm using schoolbook multiplication on the classical arithmetic logic unit (ALU) and the parallel approach on the 2-way SIMD vector instruction set of both the x86 (SSE2) and the ARM (NEON) processors. Our experimental results show that on both 32-bit x86 and ARM platforms, widely available in a broad range of mobile devices, this parallel approach manages to outperform our classical sequential implementation.

Note, that the approach and implementation used in the GNU multiple precision arithmetic library (GMP) [13], is faster than the one presented in this paper and the one used in OpenSSL [32] on some Intel platforms we tested. This approach does not use the interleaved Montgomery multiplication but first computes the multiplication, using asymptotically fast method like Karatsuba [25], followed by the Montgomery reduction. GMP uses dedicated squaring code which is not used in our implementation. Note, however, that GMP is not a cryptographic library and does not strive to provide constant-time implementations. See Section 3.1 for a more detailed discussion of the different approaches.

2 Preliminaries

In this section we recall some of the facts related to SIMD instructions and Montgomery multiplication. In Section 2.3 we summarize related work of parallel software implementations of Montgomery multiplication.

2.1 SIMD Instruction Set Extensions

Many processors include instruction set extensions. In this work we mainly focus on extensions which support vector instructions following the single instruction, multiple data (SIMD) paradigm. The two platforms we consider are the x86 and the ARM, and the instruction set extensions for these platforms are outlined below. The main vector instructions used in this work (on both processor types) are integer multiply, shift, bitwise AND, addition, and subtraction.

The x86 SIMD Instruction Set Extensions. SIMD operations on x86 and x64 processors have been supported in a number of instruction set extensions, beginning with MMX in 1997. This work uses the streaming SIMD extensions 2 (SSE2) instructions, introduced in 2001. SSE2 has been included on most Intel and AMD processors manufactured since then. We use "SSE" to refer to SSE2. SSE provides 128-bit SIMD registers (eight registers on x86 and sixteen registers on x64) which may be viewed as vectors of 1-, 8-, 16-, 32-, or 64-bit integer elements operating using 128-, 16-, 8-, 4-, or 2-way SIMD respectively. Vector operations allow multiple arithmetic operations to be performed simultaneously, for example `PMULLUDQ` multiplies the low 32-bits of a pair of 64-bit integers and outputs a pair of 64-bit integers. For a description of SSE instructions, see [22].

The ARM NEON SIMD Engine. Some ARM processors provide a set of additional SIMD operations, called NEON. The NEON register file can be viewed as either sixteen

Algorithm 1 The radix-r interleaved Montgomery multiplication [30] method.

```
Input: { $A, B, M, \mu$ such that $A = \sum_{i=0}^{n-1} a_i r^i, 0 \leq a_i < r,\ 0 \leq A, B < M,\ 2 \nmid M,$
       {                         $r^{n-1} \leq M < r^n,\ \gcd(r, M) = 1,\ \mu = -M^{-1} \bmod r.$
Output: $C \equiv A \cdot B \cdot r^{-n} \bmod M$ such that $0 \leq C < M$.
 1: $C \leftarrow 0$
 2: for $i = 0$ to $n - 1$ do
 3:     $C \leftarrow C + a_i \cdot B$
 4:     $q \leftarrow \mu \cdot C \bmod r$
 5:     $C \leftarrow (C + q \cdot M)/r$
 6: if $C \geq M$ then
 7:     $C \leftarrow C - M$
 8: return $C$
```

128-bit registers or 32 64-bit registers. The NEON registers can contain integer vectors, as in SSE. The operations provided by NEON are comparable to those provided by SSE. For example, the vector multiply instruction `vmul` takes two pairs of 32-bit integers as input and produces a pair of 64-bit outputs. This is equivalent to the SSE2 instruction `PMULUDQ`, except the inputs are provided in 64-bit registers, rather than 128-bit registers. Another example, but without an SSE equivalent is the `vmlal` instruction which performs a `vmull` and adds the results to a 128-bit register (treated as two 64-bit integers). For a complete description of the NEON instructions, see [3].

2.2 Montgomery Arithmetic

Montgomery arithmetic [30] consists of transforming operands into a Montgomery representation, performing the desired computations on these transformed numbers, then converting the result (also in Montgomery representation) back to the regular representation. Due to the overhead of changing representations, Montgomery arithmetic is best when used to replace a sequence of modular multiplications, since the overhead is amortized.

The idea behind Montgomery multiplication is to replace the expensive division operations required when computing the modular reduction by cheap shift operations (division by powers of two). Let w denote the word size in bits. We write integers in a radix r system, for $r = 2^w$ where typical values of w are $w = 32$ or $w = 64$. Let M be an n-word odd modulus such that $r^{n-1} \leq M < r^n$. The Montgomery radix r^n is a constant such that $\gcd(r^n, M) = 1$. The Montgomery residue of an integer $A \in \mathbf{Z}/M\mathbf{Z}$ is defined as $\widetilde{A} = A \cdot r^n \bmod M$. The Montgomery product of two residues is defined as $M(\widetilde{A}, \widetilde{B}) = \widetilde{A} \cdot \widetilde{B} \cdot r^{-n} \bmod M$. Algorithm 1 outlines interleaved Montgomery multiplication, denoted as coarsely integrated operand scanning in [26], where the multiplication and reduction are interleaved. Note that residues may be added and subtracted using regular modular algorithms since $\widetilde{A} \pm \widetilde{B} \equiv (A \cdot r^n) \pm (B \cdot r^n) \equiv (A \pm B) \cdot r^n \pmod{M}$.

2.3 Related Work

There has been a considerable amount of work related to SIMD implementations of cryptography. The authors of [6, 12, 35] propose ways to speed up cryptography using the NEON vector instructions. Intel's SSE2 vector instruction set extension is used to compute pairings in [15] and multiply big numbers in [21]. Simultaneously, people have studied techniques to create hardware and software implementations of Montgomery multiplication. We now summarize some of the techniques to implement Montgomery multiplication concurrently in a software implementation. A parallel software approach describing systolic (a specific arrangement of processing units used in parallel computations) Montgomery multiplication is described in [10, 23]. An approach using the vector instructions on the Cell microprocessor is considered in [8]. Exploiting much larger parallelism using the single instruction multiple threads paradigm, is realized by using a residue number system [14, 29] as described in [4]. This approach is implemented for the massively parallel graphics processing units in [19]. An approach based on Montgomery multiplication which allows one to split the operand into two parts, which can be processed in parallel, is called bipartite modular multiplication and is introduced in [24]. More recently, the authors of [18] describe an approach using the soon to be released AVX2 SIMD instructions, for Intel's Haswell architecture, which uses 256-bit wide vector instructions. The main difference between the method proposed in this work and most of the SIMD approaches referred to here is that we do not follow the approach described in [21]. We do not use a redundant representation to accumulate multiple multiplications. We use a different approach to make sure no extra words are required for the intermediate values (see Section 3).

Another approach is to use the SIMD vector instructions to compute *multiple* Montgomery multiplications in parallel. This can be useful in applications where many computations need to be processed in parallel such as batch-RSA. This approach is studied in [33] using the SSE2 vector instructions on an Pentium 4 and in [7] on the Cell processor.

3 Montgomery Multiplication using SIMD Extensions

Montgomery multiplication, as outlined in Algorithm 1, does not lend itself to parallelization directly. In this section we describe an algorithm capable of computing the Montgomery multiplication using two threads running in parallel which perform identical arithmetic steps. Hence, this algorithm can be implemented efficiently using common 2-way SIMD vector instructions. For illustrative purposes we assume a radix-2^{32} system, but this can be adjusted accordingly to other choices of radix.

As can be seen from Algorithm 1 there are two $1 \times n \to (n+1)$ limb (a_iB and qM) and a single $1 \times 1 \to 1$ limb ($\mu C \bmod r$) multiplications per iteration. These three multiplications depend on each other, preventing concurrent computation. In order to remove this dependence, note that for the computation of q only the first limb c_0 of $C = \sum_{i=0}^{n-1} c_i 2^{32i}$ is required. Hence, if one is willing to compute the updated value of c_0 twice then the two larger $1 \times n \to (n+1)$ limb multiplications become independent of each other and can be computed in parallel. More precisely, lines 3, 4, and 5 of Algorithm 1 can be replaced with

$$q \leftarrow ((c_0 + a_i \cdot b_0)\mu) \bmod r$$
$$C \leftarrow (C + a_i \cdot B + q \cdot M)/r$$

Algorithm 2 A parallel radix-2^{32} interleaved Montgomery multiplication algorithm. Except for the computation of q, the arithmetic steps in the outer for-loop performed by both Computation 1 and Computation 2 are identical. This approach is suitable for 32-bit 2-way SIMD vector instruction units. Note that the value of the precomputed Montgomery inverse μ is different ($\mu = M^{-1} \bmod 2^{32}$) than the one used in Algorithm 1 ($\mu = -M^{-1} \bmod 2^{32}$).

Input: $\begin{cases} A, B, M, \mu \text{ such that } A = \sum_{i=0}^{n-1} a_i 2^{32i}, B = \sum_{i=0}^{n-1} b_i 2^{32i}, \\ M = \sum_{i=0}^{n-1} m_i 2^{32i}, 0 \le a_i, b_i < 2^{32},\ 0 \le A, B < M, \\ 2^{32(n-1)} \le M < 2^{32n}, 2 \nmid M,\ \mu = M^{-1} \bmod 2^{32}. \end{cases}$

Output: $C \equiv A \cdot B \cdot 2^{-32n} \bmod M$ such that $0 \le C < M$.

```
Computation 1                                   | Computation 2
$d_i = 0$ for $0 \le i < n$                     | $e_i = 0$ for $0 \le i < n$
for $j = 0$ to $n-1$ do                         | for $j = 0$ to $n-1$ do
                                                |   $q \leftarrow ((\mu \cdot b_0) \cdot a_j + \mu \cdot (d_0 - e_0)) \bmod 2^{32}$
  $t_0 \leftarrow a_j \cdot b_0 + d_0$          |   $t_1 \leftarrow q \cdot m_0 + e_0$ // Note that $t_0 \equiv t_1 \pmod{2^{32}}$
  $t_0 \leftarrow \lfloor t_0 / 2^{32} \rfloor$ |   $t_1 \leftarrow \lfloor t_1 / 2^{32} \rfloor$
  for $i = 1$ to $n-1$ do                       |   for $i = 1$ to $n-1$ do
    $p_0 \leftarrow a_j \cdot b_i + t_0 + d_i$  |     $p_1 \leftarrow q \cdot m_i + t_1 + e_i$
    $t_0 \leftarrow \lfloor p_0 / 2^{32} \rfloor$ |   $t_1 \leftarrow \lfloor p_1 / 2^{32} \rfloor$
    $d_{i-1} \leftarrow p_0 \bmod 2^{32}$       |     $e_{i-1} \leftarrow p_1 \bmod 2^{32}$
  $d_{n-1} \leftarrow t_0$                      |   $e_{n-1} \leftarrow t_1$
            $\searrow$                          |          $\swarrow$
        $C \leftarrow D - E$   // where $D = \sum_{i=0}^{n-1} d_i 2^{32i}$, $E = \sum_{i=0}^{n-1} e_i 2^{32i}$
        if $C < 0$ do $C \leftarrow C + M$
```

ensuring that the two larger multiplications do not depend on each other.

The second idea is to flip the sign of the Montgomery constant μ: i.e. instead of using $-M^{-1} \bmod 2^{32}$ (as in Algorithm 1) we use $\mu = M^{-1} \bmod 2^{32}$ (the reason for this choice is outlined below). When computing the Montgomery product $C = A \cdot B \cdot 2^{-32n} \bmod M$, for an odd modulus M such that $2^{32(n-1)} \le M < 2^{32n}$, one can compute D, which contains the sum of the products $a_i B$, and E, which contains the sum of the products qM, separately. Due to our choice of the Montgomery constant μ we have $C = D - E \equiv A \cdot B \cdot 2^{-32n} \pmod{M}$, where $0 \le D, E < M$: the maximum values of both D and E fit in an n-limb integer, avoiding a carry that might result in an $(n+1)$ limb long integer as in Algorithm 1. This approach is outlined in Algorithm 2.

At the start of every iteration of j the two separate computations need some communication in order to compute the new value of q. In practice, this communication requires extracting the values d_0 and e_0, the first limb of D and E respectively, from the SIMD vector. No such extracting is required in the inner-most loop over the i values in Algorithm 2. The value of q is computed as

$$q = ((\mu \cdot b_0) \cdot a_j + \mu \cdot (d_0 - e_0)) \bmod 2^{32} = \mu(a_j \cdot b_0 + c_0) \bmod 2^{32}$$

since $c_0 = d_0 - e_0$. Note that one can compute $(\mu \cdot b_0) \bmod 2^{32}$ at the beginning of the algorithm once and reuse it for every iteration of the for-loop.

Table 1. A simplified comparison, only stating *the number of arithmetic operations required*, of the expected performance of Montgomery multiplication when using a $32n$-bit modulus for a positive even integer n. The left side of the table shows arithmetic instruction counts for the sequential algorithm using the classical ALU (Algorithm 1) and when using 2-way SIMD instructions with the parallel algorithm (Algorithm 2). The right side of the table shows arithmetic instruction counts when using one level of Karatuba's method [25] for the multiplication as analyzed in [17].

Instruction	classical		2-way SIMD	Karatsuba	Instruction
	32-bit	64-bit	32-bit	32-bit	
`add`	-	-	n	$\frac{13}{4}n^2 + 8n + 2$	`add`
`sub`	-	-	n	$\frac{7}{4}n^2 + n$	`mul`
`shortmul`	n	$\frac{n}{2}$	$2n$		
`muladd`	$2n$	n	-		
`muladdadd`	$2n(n-1)$	$n(\frac{n}{2}-1)$	-		
`SIMD muladd`	-	-	n		
`SIMD muladdadd`	-	-	$n(n-1)$		

Except for the computation of q, all arithmetic computations performed by Computation 1 and Computation 2 are identical but work on different data. This makes Algorithm 2 suitable for implementation using 2-way 32-bit SIMD vector instructions. This approach benefits from 2-way SIMD $32 \times 32 \rightarrow 64$-bit multiplication and matches exactly the 128-bit wide vector instructions as present in SSE and NEON. Changing the radix used in Algorithm 2 allows implementation with larger or smaller vector instructions. For example, if a $64 \times 64 \rightarrow 128$-bit vector multiply instruction is provided in a future version of AVX, implementing Algorithm 2 in a 2^{64}-radix system with 256-bit wide vector instructions could potentially speed-up modular multiplication by a factor of up to two on 64-bit systems (see Section 3.1).

At the end of Algorithm 2, there is a conditional addition, as opposed to a conditional subtraction in Algorithm 1, due to our choice of μ. The condition is whether $D - E$ is negative (produces a borrow), in this case the modulus must be added to make the result positive. This conditional addition can be converted into straight-line code by creating a mask depending on the borrow and selecting either $D - E$ (if there is no borrow) or $D - E + M$ (if there is a borrow) so that the code runs in constant-time (an important characteristic for side-channel resistance [27]).

3.1 Expected Performance

The question remains if Algorithm 2, implemented for a 2-way SIMD unit, outperforms Algorithm 1, implemented for the classical ALU. This mainly depends on the size of the inputs and outputs of the integer instructions, how many instructions can be dispatched per cycle, and the number of cycles an instruction needs to complete. In order to give a (simplified) prediction of the performance we compute the expected performance of a Montgomery multiplication using a $32n$-bit modulus for a positive even integer n. Let $\texttt{muladd}_w(e, a, b, c)$ and $\texttt{muladdadd}_w(e, a, b, c, d)$ denote the computation of $e = a \times b + c$ and $e = a \times b + c + d$, respectively, for $0 \leq a, b, c, d < 2^w$ and $0 \leq e < 2^{2w}$ as a basic operation on a compute architecture which works on w-bit words. Some platforms have these operations as a single instruction (e.g., on some ARM architectures) or they must be implemented using a multiplication and addition(s) (as on the x86 platform). Furthermore, let $\texttt{shortmul}_w(e, a, b)$ denote $e = a \times b \bmod 2^w$: this only computes the lower word of the result and can be done faster (compared to a full product) on most platforms.

Table 1 summarizes the expected performance of Algorithm 1 and 2 in terms of arithmetic operations only (e.g., the data movement, shifting and masking operations are omitted). Also the operations required to compute the final conditional subtraction or addition have been omitted. When solely considering the `muladd` and `muladdadd` instructions it becomes clear from Table 1 that the SIMD approach uses exactly half of the number of operations compared to the 32-bit classical implementation and almost twice as many operations compared to the classical 64-bit implementations. However, the SIMD approach requires more operations to compute the value of q every iteration and has various other overhead (e.g., inserting and extracting values from the vector). Hence, when assuming that all the characteristics of the SIMD and classical (non-SIMD) instructions are identical, which will not be the case on all platforms, then we expect Algorithm 2 running on a 2-way 32-bit SIMD unit to outperform a classical 32-bit implementation using Algorithm 1 by at most a factor of two while being roughly twice as slow when compared to a classical 64-bit implementation.

Inherently, the interleaved Montgomery multiplication algorithm (as used in this work) is not compatible with asymptotically faster integer multiplication algorithms like Karatsuba multiplication [25]. We have not implemented the Montgomery multiplication by first computing the multiplication using such faster methods, and then computing the modular reduction, using SIMD vector instructions in one or both steps. In [17], instruction counts are presented when using the interleaved Montgomery multiplication, as used in our baseline implementation, as well as for an approach where the multiplication and reduction are computed separately. Separating these two steps makes it easier to use a squaring algorithm. In [17] a single level of Karatsuba on top of Comba's method [9] is considered: the arithmetic instruction counts are stated in Table 1. For 1024-bit modular multiplication (used for 2048-bit RSA decryption using the CRT), the Karatsuba approach can reduce the number of multiplication and addition instructions by a factor 1.14 and 1.18 respectively on 32-bit platforms compared to the sequential interleaved approach. When comparing the arithmetic instructions only, the SIMD approach for interleaved Montgomery multiplication is 1.70 and 1.67 times faster than the sequential Karatsuba approach for 1024-bit modular multiplication on 32-bit platforms. Obviously, the Karatsuba approach can be sped up using SIMD instructions as well.

The results in Table 1 are for Montgomery multiplication only. It is known how to optimize (sequential) Montgomery squaring [16], but as far as we are aware, not how to optimize squaring using SIMD instructions. Following the analysis from [17], the cost of a Montgomery squaring is $\frac{11n+14}{14n+8}$ and $\frac{3n+5}{4n+2}$ the cost of a Montgomery multiplication when using the Karatsuba or interleaved Montgomery approach on n-limb integers. For 1024-bit modular arithmetic (as used in RSA-2048 with $n = 32$) this results in 0.80 (for Karatsuba) and 0.78 (for interleaved). For RSA-2048, approximately 5/6 of all operations are squarings: this highlights the potential of an efficient squaring implementation.

4 Implementation Results

We have implemented interleaved Montgomery modular multiplication (Algorithm 1) as a baseline for comparison with the SIMD version (Algorithm 2). In both implementations, the final addition/subtraction was implemented using masking such that it runs in constant time, to resist certain types of side-channel attacks using timing and branch prediction. Since the cost of this operation was observed to be a small fraction of the overall cost, we chose not to write a separate optimized implementation for operations using only public values (such as signature verification).

Benchmark Hardware. Our implementations were benchmarked on recent Intel x86-32, x64 and ARM platforms. On the Intel systems, Windows 7 and Windows 8 were used, and on ARM systems Windows RT was used. The Microsoft C/C++ Optimizing Compiler Version 16.10 was used for x86 and x64, and version 17.00 was used for ARM.[1] Our benchmark systems are the following:

Intel Xeon E31230 A quad core 3.2 GHz CPU on an HP Z210 workstation. We used SSE2 for Algorithm 2 and also benchmark x86-32 and x86-64 implementations of Algorithm 1 for comparison.

Intel Atom Z2760 A dual core 1.8 GHz system-on-a-chip (SoC), on an Asus Vivo Tab Smart Windows 8 tablet.

NVIDIA Tegra T30 A quad core 1.4 GHz ARM Cortex-A9 SoC, on an NVIDIA developer tablet.

Qualcomm MSM8960 A quad core 1.8 GHz Snapdragon S4 SoC, on a Dell XPS 10 tablet.

NVIDIA Tegra 4 A quad core 1.91 GHz ARM Cortex-A15 SoC, on an NVIDIA developer tablet.

We chose to include the Xeon processor to confirm the analysis of Section 3.1, that the x64 implementation should give the best performance, and to compare it with the SIMD implementation. The other processors are common in tablets and smartphones, and on these platforms, the SIMD implementation should be the best available. The performance of 32-bit code is also of interest on 64-bit systems, since 32-bit crypto libraries are included on 64-bit systems (e.g., on 64-bit Windows), to allow existing x86 applications to run on the 64-bit system without being ported and recompiled.

On the Xeon system, Intel's *Turbo Boost* feature will dynamically increase the frequency of the processor under high computational load. We found Turbo Boost had a modest impact on our timings. Since it is a potential source of variability, all times reported here were measured with Turbo Boost disabled.

[1] These were the newest versions available for each architecture at the time of writing.

Benchmarks. We chose to benchmark the cost of modular multiplication for 512-bit, 1024-bit and 2048-bit moduli, since these are currently used in deployed cryptography. The 512-bit modular multiplication results may also be interesting for usage in elliptic curve and pairing based cryptosystems. We created implementations optimized for these "special" bitlengths as well as *generic* implementations, i.e., implementations that operate with arbitrary length inputs. For comparison, we include the time for modular multiplication with 1024- and 2048-bit generic implementations. Our x64 baseline implementation has no length-specific code (we did not observe performance improvements).

We also benchmark the cost of RSA encryption and decryption using the different modular multiplication routines. We do not describe our RSA implementation in detail, because it is the same for all benchmarks, but note that: (i) decryption with an n-bit modulus is done with $n/2$-bit arithmetic using the Chinese remainder theorem approach, (ii) this is a "raw" RSA operation, taking an integer as plaintext input, no padding is performed, (iii) no specialized squaring routine is used, and (iv) the public exponent in our benchmarks is always $2^{16} + 1$. We compute the modular exponentiation using a windowing based approach. As mentioned in (iii), we have not considered a specialized Montgomery squaring algorithm for the sequential or the SIMD algorithms. Using squaring routines can significantly enhance the performance of our code as discussed in 3.1.

All of our benchmarks are *average* times, computed over 10^5 runs for modular multiplication, and 100 runs for RSA operations, with random inputs for each run. With these choices the standard deviation is three percent or less. Note that the performance results for RSA-1024 are stated for comparison's sake only, this 80-bit secure scheme should not be used anymore (see NIST SP 800-57 [31]).

x86/x64 Results. In the first 32-bit benchmark (Xeon x86), our implementation using SIMD instructions is 1.6 to 2.5 times faster than the serial version (see Table 2). The speed-up of the length-specific SIMD implementation over the generic implementation is on average a factor 1.4, noticeably more than the factor 1.2 for the baseline. Algorithm 2 results in faster RSA operations as well, which are roughly sped-up by a factor of two. Our second 32-bit benchmark (Atom x86) was on average 1.69 times faster than our baseline. This makes our SIMD algorithm the better option on this platform. However, the speed-up observed was not as large as our Xeon x86 benchmark. This may be because the Atom has an in-order instruction scheduler. The 64-bit implementation of Algorithm 1 is roughly four times as fast as the 32-bit implementation and the SIMD algorithm (also 32-bit) is right in the middle, roughly twice as slow as the 64-bit algorithm. This agrees with our analysis from Section 3.1. On all platforms the performance ratio between baseline and SIMD is slightly worse for 512-bit moduli due to the overhead of using SIMD instructions. Algorithm 2 is still faster than the baseline for 512-bit moduli on the Xeon x86, Atom and the Snapdragon S4.

ARM Results. On ARM our results are more mixed (see Table 3). First we note that on the Tegra 3 SoC, our NEON implementation of Algorithm 2 is consistently worse than the baseline, almost twice as slow. Going back to our analysis in Section 3.1, this would occur if the cost of a vector multiply instruction (performing two 32-bit multiplies) was about

Table 2. Implementation timings in microseconds and cycles for x86/x64 based processors. The "ratio" column is baseline/SIMD. The 512**g**, 1024**g** and 2048**g** rows are generic implementations that do not optimize for a specific bitlength.

Benchmark	Xeon x86			Xeon x64			Atom (x86)		
	baseline	SIMD	ratio	baseline	SIMD	ratio	baseline	SIMD	ratio
modmul 512	0.906	0.492	1.81	0.221	0.492	0.45	4.934	2.864	1.72
(cycles)	2899	1577	1.83	711	1577	0.45	8888	5168	1.72
modmul 1024	3.126	1.407	2.22	0.589	1.407	0.42	17.428	10.550	1.65
(cycles)	9989	4500	2.22	1886	4500	0.42	31342	18984	1.65
RSA enc 1024	75.459	36.745	2.05	16.411	36.745	0.45	407.835	250.285	1.63
(cycles)	241014	117419	2.05	52457	117419	0.45	733224	450092	1.63
RSA dec 1024	1275.030	656.831	1.94	278.444	656.831	0.42	6770.646	4257.838	1.59
(cycles)	4070962	2097258	1.94	889103	2097258	0.42	12167933	7652178	1.59
modmul 2048	13.371	5.000	2.67	2.263	5.000	0.45	72.320	36.880	1.96
(cycles)	42698	15972	2.67	7230	15972	0.45	129994	66299	1.96
RSA enc 2048	277.719	129.876	2.14	56.813	129.876	0.44	1437.459	891.185	1.61
(cycles)	886828	414787	2.14	181412	414787	0.44	2583643	1601878	1.61
RSA dec 2048	8231.233	3824.690	2.15	1543.666	3824.690	0.40	44629.140	28935.088	1.54
(cycles)	26280725	12211700	2.15	4928633	12211700	0.40	80204317	52000367	1.54
modmul 512**g**	1.011	0.606	1.67	0.221	0.606	0.36	5.309	3.376	1.57
(cycles)	3234	1941	1.67	711	1941	0.37	9561	6087	1.57
modmul 1024**g**	3.760	2.222	1.69	0.734	2.222	0.33	18.962	11.475	1.65
(cycles)	12012	7101	1.69	2350	7101	0.33	34099	20644	1.65
modmul 2048**g**	13.326	8.968	1.49	2.733	8.968	0.30	73.898	42.901	1.72
(cycles)	42555	28640	1.49	8734	28640	0.30	132831	77125	1.72

the cost of two non-vector multiply instructions. This is (almost) the case according to the Cortex-A9 instruction latencies published by ARM.[2] Our efforts to pipeline multiple vector multiply instructions did not sufficiently pay off – the length-specific implementations give a 1.27 factor speed-up over the generic implementations, roughly the same speed-up obtained when we optimize the baseline for a given bitlength (by fully unrolling the inner loop).

On the newer ARM SoCs in our experiments, the S4 and Tegra 4, the results are better. On the Snapdragon S4 the SIMD implementation is consistently better than the baseline. The NEON length-specific implementations were especially important and resulted in a speed-up by a factor of 1.30 to 1.40 over generic implementations, while optimizing the baseline implementation for a specific length was only faster by a factor slightly above 1.10. This is likely due to the inability of the processor to effectively re-order NEON instructions to minimize pipeline stalls – the main difference in our length-specific implementation was to partially unroll the inner loop and re-order instructions to use more registers and pipeline four multiply operations.

Performance of the SIMD algorithm on the Tegra 4 was essentially the same as the baseline performance. This is a solid improvement in NEON performance compared to our benchmarks on the Tegra 3, however the Tegra 4's NEON performance still lags behind the S4 (for the instructions used in our benchmarks). We suspect (based on informal

[2] Results of the NEON `vmull`/`vmlal` instructions are available after 7 cycles, while the two 32-bit outputs of the ARM `umaal` instruction become ready after 4 and 5 cycles [1, 2].

Table 3. Implementation timings in microseconds for ARM-based processors. The "ratio" column is baseline/SIMD. The 512**g**, 1024**g** and 2048**g** rows are generic implementations that do not optimize for a specific bitlength.

Benchmark	Snapdragon S4			Tegra 4			Tegra 3		
	baseline	SIMD	ratio	baseline	SIMD	ratio	baseline	SIMD	ratio
modmul 512	2.630	1.888	1.39	1.272	1.328	0.96	2.393	4.005	0.60
(cycles)	4199	3079	1.36	2373	2473	0.96	3175	5236	0.61
modmul 1024	9.127	5.452	1.67	4.709	4.624	1.02	7.854	13.556	0.58
(cycles)	14041	8467	1.66	8681	8527	1.02	10167	17464	0.58
RSA enc 1024	198.187	142.956	1.38	168.617	179.227	0.94	189.420	295.110	0.64
(cycles)	302898	219244	1.38	195212	207647	0.94	245167	379736	0.65
RSA dec 1024	3424.413	2475.716	1.38	1999.211	2303.588	0.87	3306.230	5597.280	0.59
(cycles)	5179365	3746371	1.38	3288177	3332262	0.99	4233862	7166897	0.59
modmul 2048	33.987	18.599	1.82	18.498	18.200	1.02	28.611	49.825	0.57
(cycles)	51623	28356	1.82	33961	33441	1.02	36746	63900	0.57
RSA enc 2048	716.160	467.713	1.53	593.326	617.758	0.96	679.920	1060.050	0.64
(cycles)	1087318	710910	1.53	725336	712542	1.02	872468	1358955	0.64
RSA dec 2048	22992.576	14202.886	1.62	19024.405	19797.988	0.96	21519.880	36871.550	0.58
(cycles)	34769147	21478047	1.62	23177617	22812040	1.02	27547434	47205919	0.58
modmul 512**g**	3.147	2.510	1.25	1.493	1.467	1.02	2.990	4.916	0.61
(cycles)	4988	4018	1.24	2777	2736	1.01	3937	6404	0.61
modmul 1024**g**	10.600	7.885	1.34	5.062	6.937	0.73	10.385	16.991	0.61
(cycles)	16267	12155	1.34	9327	12777	0.73	13408	21870	0.61
modmul 2048**g**	38.094	27.913	1.36	19.137	18.200	1.05	36.839	63.218	0.58
(cycles)	57833	42441	1.36	35139	33441	1.05	47279	81048	0.58

experiments) that an implementation of Algorithm 2 specifically optimized for the Tegra 4 could significantly outperform the baseline, but still would not be comparable to the S4.

There is a slight difference between the cycle count measurement and the microsecond measurement for the 2048-bit ARM benchmarks on the Tegra 4. To measure cycles on ARM we read the cycle count register (`PMCCNTR`), and time is measured with the Windows `QueryPerformanceCounter` function. Since these are different time sources, a small difference is not surprising.

4.1 Comparison to Previous Work

Comparison to eBACS and OpenSSL. We have compared our SIMD implementation of the interleaved Montgomery multiplication algorithm to our baseline implementation of this method. To show that our baseline is competitive and put our results in a wider context, we compare to benchmark results from eBACS: ECRYPT Benchmarking of Cryptographic Systems [5] and to OpenSSL [32]. Table 4 summarizes the cycle counts from eBACS on platforms which are close to the ones we consider in this work, and also includes the results of running the performance benchmark of OpenSSL 1.0.1e [32] on our Atom device. As can be seen from Table 4, our baseline implementation results from Table 2 and Table 3 are similar (except for 1024-bit RSA decryption, which our implementation does not optimize, as mentioned above).

Table 4. Performance results expressed in cycles of RSA 1024-bit and 2048-bit encryption (enc) and decryption (dec). The first four performance numbers have been obtained from eBACS: ECRYPT Benchmarking of Cryptographic Systems [5] while the fifth row corresponds to running the performance benchmark suite of OpenSSL [32] on the same Atom device used to obtain the performance results in Table 2. The last two rows correspond to running GMP on our Atom and Xeon (in 32-bit mode).

Platform	RSA 1024		RSA 2048	
	Enc	Dec	Enc	Dec
ARM – Tegra 250 (1000 MHz)	261677	11684675	665195	65650103
ARM – Snapdragon S3 (1782 MHz)	276836	7373869	609593	39746105
x86 – Atom N280 (1667 MHz)	315620	13116020	871810	81628170
x64 – Xeon E3-1225 (3100 MHz)	49652	1403884	103744	6158336
x86 – Atom Z2760 (1800 MHz)	610200	10929600	2323800	75871800
x86 – Atom Z2760 (1800 MHz)	305545	5775125	2184436	37070875
x86 – Xeon E3-1230 (3200 MHz)	106035	1946434	695861	11929868

Comparison to GMP. The implementation in the GNU multiple precision arithmetic library (GMP) [13] is based on the non-interleaved Montgomery multiplication. This means the multiplication is computed first, possibly using a asymptotically faster algorithm than schoolbook, followed by the Montgomery reduction (see Section 3.1). The last two rows in Table 4 summarize performance numbers for our Atom and Xeon (in 32-bit mode) platforms. The GMP performance numbers for RSA-2048 decryption on the Atom (37.1 million) are significantly faster compared to OpenSSL (75.9 million), our baseline (80.2 million) and our SIMD (52.0 million) implementations. On the 32-bit Xeon the performance of the GMP implementation, which uses SIMD instructions for the multiplication and has support for asymptotically faster multiplication algorithms, is almost identical to our SIMD implementation which uses interleaved Montgomery multiplication. Note that both OpenSSL and our implementations are designed to resist side-channel attacks, and run in constant time, while both the GMP modular exponentiation and multiplication are not, making GMP unsuitable for use in many cryptographic applications. The multiplication and reduction routines in GMP can be adapted for cryptographic purposes but it is unclear at what performance price. From Table 2, it is clear that our SIMD implementation performs better on the 32-bit Xeon than on the Atom. The major difference between these two processors is the instruction scheduler (in-order on the Atom and out-of-order on the Xeon).

4.2 Engineering Challenges

In this section we discuss some engineering challenges we had to overcome in order to use SIMD in practice. Our goal is an implementation that is efficient and supports multiple processors, but is also maintainable. The discussion here may not be applicable in other circumstances.

ASM or intrinsics? There are essentially two ways to access the SIMD instructions directly from a C program. One either writes assembly language (ASM), or uses compiler intrinsics. Intrinsics are macros that the compiler translates to specific instructions, e.g., on ARM, the Windows RT header file `arm_neon.h` defines the intrinsic `vmull_u32`, which the compiler implements with the `vmull` instruction. In addition to instructions, the header also exposes special data types corresponding to the 64 and 128-bit SIMD registers. We chose to use intrinsics for our implementation, for the following reasons. C with intrinsics is easier to debug, e.g., it is easier to detect mistakes using assertions. Furthermore, while there is a performance advantage for ASM implementations, these gains are limited in comparison to a careful C implementation with intrinsics (in our experience). In addition ASM is difficult to maintain. For example, in ASM the programmer must handle all argument passing and set up the stack frame, and this depends on the calling conventions. If calling conventions are changed, the ASM will need to be rewritten, rather than simply recompiled. Also, when writing for the Microsoft Visual Studio Compiler, the compiler automatically generates the code to perform structured exception handling (SEH), which is an exception handling mechanism at the system level for Windows and a requirement for all code running on this operating system. Incorrect implementation of SEH code may result in security bugs that are often difficult to detect until they are used in an exploit. Also, compiler features such as Whole Program Optimization and Link Time Code generation, that optimize code layout and time-memory usage tradeoffs, will not work correctly on ASM.

Despite the fact that one gets more control of the code (e.g. register usage) when writing in ASM, using instrinsics and C can still be efficient. Specifically, we reviewed the assembly code generated by the compiler to ensure that the run-time of this code remains in constant time and register usage is as we expected. In short, we have found that ASM implementations require increased engineering time and effort, both in initial development and maintenance, for a relatively small gain in performance. We have judged that this trade off is not worthwhile for our implementation.

`simd.h` abstraction layer Both SSE2 and NEON vector instructions are accessible as intrinsics, however, the types and instructions available for each differ. To allow a single SIMD implementation to run on both architectures, we abstracted a useful subset of SSE2 and NEON in header named `simd.h`. Based on the architecture, this header defines inline functions wrapping a processor-specific intrinsic. `simd.h` also refines the vector data types, e.g., the type `simd32x2p_t` stores two 32-bit unsigned integers in a 64-bit register on ARM, but on x86 stores them in a 128-bit integer (in bits 0–31 and 64–95), so that they are in the correct format for the vector multiply instruction (which returns a value of type `simd64x2_t` on both architectures). The compiler will check that the arguments to the `simd.h` functions match the prototype, something that is not possible with intrinsics (which are preprocessor macros). While abstraction layers are almost always technically possible, we find it noteworthy that in this case it can be done without adding significant overhead, and code using the abstraction performs well on multiple processors. With `simd.h` containing all of architecture-specific code, the SIMD timings in the tables above were generated with two implementations: a generic one, and a length-specific one that requires the number of limbs in the modulus be divisible by four, to allow partial unrolling of the inner loop of Algorithm 2.

Length-specific routines. Given the results from Table 2 and Table 3, it is clear that having specialized routines for certain bitlengths is worthwhile. In a math library used to implement multiple crypto primitives, each supporting a range of allowed keysizes, routines for arbitrary length moduli are required as well. This raises the question of how to automatically select one of multiple implementations. We experimented with two different designs. The first design stores a function pointer to the modular multiplication routine along with the modulus. The second uses a function pointer to a length-specific exponentiation routine. On the x86 and x64 platforms, with 1024-bit (and larger) operands, the performance difference between the two approaches is small (the latter was faster by a factor around 1.10), however on ARM, using function pointers to multiplication routines is slower by a factor of up to 1.30 than when using pointers to exponentiation routines. The drawback of this latter approach is the need to maintain multiple exponentiation routines.

SoC-specific routines. Our experiments with multiple ARM SoCs also show that performance can vary by SoC. This is expected, however we were surprised by the range observed, compared to x86/x64 processors which are more homogeneous. We also observed that small code changes can result in simultaneous speed improvements on one SoC, and regression on another. Our current implementation performs a run-time check to identify the SoC, to decide whether to use Algorithm 1 or 2. Our results highlight that there is a great deal of variability between different implementations of the ARM architecture and that, for the time being, it is difficult to write code that performs well on multiple ARM SoCs simultaneously. This also implies that published implementation results for one ARM microprocessor core give little to no information on how it would perform on another. For more information, see the ARM technical reference manuals [3].

5 Conclusions and Future Work

In this paper we present a parallel version of the interleaved Montgomery multiplication algorithm that is amenable to implementation using widely available SIMD vector extension instructions (SSE2 and NEON). The practical implications of this approach are highlighted by our performance results on common tablet devices. When using 2048-bit moduli we are able to outperform our sequential implementation using the schoolbook multiplication method by a factor of 1.68 to 1.76 on both 32-bit x86 and ARM processors.

The performance numbers agree with our analysis that a 2-way SIMD implementation using 32-bit multipliers is not able to outperform a classical interleaved Montgomery multiplication implementation using 64-bit multiplication instructions. Hence, we also conclude that it would be beneficial for new 256-bit SIMD instruction sets to include 2-way integer multipliers. For example, our results suggest that modular multiplication could be sped-up by up to a factor of two on x64 systems if a future set of AVX instructions included a $64 \times 64 \rightarrow 128$-bit 2-way SIMD multiplier.

It remains of independent interest to study ways to use both asymptotically faster integer multiplication methods (like Karatsuba) and Montgomery reduction using SIMD instructions to reduce latency, including side-channel protections. This is left as future work. Furthermore, as pointed out by an anonymous reviewer, another possibility might be to compute the proposed parallel Montgomery multiplication routine using both the integer and floating point unit instead of using vector instructions.

Acknowledgements. The authors would like to thank: Adam Glass for discussions on ARM SoCs; Patrick Longa for comments on baseline implementations and general help; Jason Mackay for catching mistakes in early drafts; Paul Schofield for help timing on the Tegra 4; and Niels Ferguson for discussions of SIMD. Also, we thank the anonymous reviewers of SAC for their helpful feedback and thank Daniel J. Bernstein and Tanja Lange for the additional suggestions, both of which improved the quality of this paper.

References

1. ARM. Cortex-A9. Technical Reference Manual, 2010. Version r2p2.
2. ARM. Cortex-A9 NEON Media Processing Engine. Technical Reference Manual, 2012. Version r4p1.
3. ARM Limited. *ARM Architechture Reference Manual ARMv7-A and ARMv7-R edition*, 2010.
4. J.-C. Bajard, L.-S. Didier, and P. Kornerup. An RNS Montgomery modular multiplication algorithm. *IEEE Trans. Computers*, 47(7):766–776, 1998.
5. D. J. Bernstein and T. Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`, accessed 2 July 2013.
6. D. J. Bernstein and P. Schwabe. NEON crypto. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES*, volume 7428 of *LNCS*, pages 320–339. Springer, 2012.
7. J. W. Bos. High-performance modular multiplication on the Cell processor. In M. A. Hasan and T. Helleseth, editors, *Arithmetic of Finite Fields – WAIFI 2010*, volume 6087 of *LNCS*, pages 7–24. Springer, Heidelberg, 2010.
8. J. W. Bos and M. E. Kaihara. Montgomery multiplication on the Cell. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics – PPAM 2009*, volume 6067 of *LNCS*, pages 477–485. Springer, Heidelberg, 2010.
9. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.
10. B. Dixon and A. K. Lenstra. Massively parallel elliptic curve factoring. In R. A. Rueppel, editor, *Eurocrypt 1992*, volume 658 of *LNCS*, pages 183–193. Springer, Heidelberg, 1993.
11. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. Blakley and D. Chaum, editors, *Crypto 1984*, volume 196 of *LNCS*, pages 10–18. Springer, Heidelberg, 1985.
12. A. Faz-Hernandez, P. Longa, and A. 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/`.
13. Free Software Foundation, Inc. *GMP: The GNU Multiple Precision Arithmetic Library*, 2013. Available at `http://www.gmplib.org/`.
14. H. L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, 8:140–147, 1959.
15. P. Grabher, J. Großschädl, and D. Page. On software parallel implementation of cryptographic pairings. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *Selected Areas in Cryptography – SAC*, volume 5381 of *LNCS*, pages 35–50. Springer, 2008.
16. J. Großschädl. Architectural support for long integer modulo arithmetic on RISC-based smart cards. *International Journal of High Performance Computer Applications – IJHPCA*, 17(2):135–146, 2003.
17. J. Großschädl, R. M. Avanzi, E. Savas, and S. Tillich. Energy-efficient software implementation of long integer modular arithmetic. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES*, volume 3659 of *LNCS*, pages 75–90. Springer, 2005.
18. S. Gueron and V. Krasnov. Software implementation of modular exponentiation, using advanced vector instructions architectures. In F. Özbudak and F. Rodríguez-Henríquez, editors, *Arithmetic of Finite Fields – WAIFI*, volume 7369 of *LNCS*, pages 119–135. Springer, 2012.
19. O. Harrison and J. Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In B. Preneel, editor, *Africacrypt 2009*, volume 5580 of *LNCS*, pages 350–367. Springer, Heidelberg, 2009.
20. R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The SSL landscape: a thorough analysis of the x.509 PKI using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 427–444. ACM, 2011.

21. Intel Corporation. Using streaming SIMD extensions (SSE2) to perform big multiplications. Whitepaper AP-941, 2000. `http://software.intel.com/file/24960`.
22. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual (Combined Volumes 1, 2A, 2B, 2C, 3A, 3B and 3C)*, 2013. Available at `http://download.intel.com/products/processor/manual/325462.pdf`.
23. K. Iwamura, T. Matsumoto, and H. Imai. Systolic-arrays for modular exponentiation using Montgomery method (extended abstract). In R. A. Rueppel, editor, *Eurocrypt*, volume 658 of *LNCS*, pages 477–481. Springer, 1992.
24. M. E. Kaihara and N. Takagi. Bipartite modular multiplication method. *IEEE Transactions on Computers*, 57(2):157–164, 2008.
25. A. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. Number 145 in Proceedings of the USSR Academy of Science, pages 293–294, 1962.
26. K. Koc, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *Micro, IEEE*, 16(3):26–33, 1996.
27. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Crypto 1996*, volume 1109 of *LNCS*, pages 104–113. Springer, Heidelberg, 1996.
28. A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Public keys. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *LNCS*, pages 626–642. Springer, 2012.
29. R. D. Merrill. Improving digital computer performance using residue number theory. *Electronic Computers, IEEE Transactions on*, EC-13(2):93–101, April 1964.
30. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
31. National Institute of Standards and Technology. Special publication 800-57: Recommendation for key management part 1: General (revision 3). `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`.
32. OpenSSL. The open source toolkit for SSL/TLS, 2013.
33. D. Page and N. P. Smart. Parallel cryptographic arithmetic using a redundant Montgomery representation. *IEEE Transactions on Computers*, 53(11):1474–1482, 2004.
34. 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.
35. A. H. Sánchez and F. Rodríguez-Henríquez. NEON implementation of an attribute-based encryption scheme. In M. J. Jacobson Jr., M. E. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *Applied Cryptography and Network Security – ACNS*, volume 7954 of *LNCS*, pages 322–338. Springer, 2013.
36. 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`.