New Speed Records for Montgomery Modular Multiplication on 8-bit AVR Microcontrollers

Zhe Liu and Johann Großschädl

University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue Richard Coudenhove-Kalergi, L–1359 Luxembourg
{zhe.liu,johann.groszschaedl}@uni.lu

Abstract. Modular multiplication of large integers is a performance-critical arithmetic operation of many public-key cryptosystems such as RSA, DSA, Diffie-Hellman (DH) and their elliptic curve-based variants ECDSA and ECDH. The computational cost of modular multiplication and related operations (e.g. exponentiation) poses a practical challenge to the widespread deployment of public-key cryptography, especially on embedded devices equipped with 8-bit processors (smart cards, wireless sensor nodes, etc.). In this paper, we describe basic software techniques to improve the performance of Montgomery modular multiplication on 8-bit AVR-based microcontrollers. First, we present a new variant of the widely-used hybrid method for multiple-precision multiplication that is 10.6% faster than the original hybrid technique of Gura et al. Then, we discuss different hybrid Montgomery multiplication algorithms, including Hybrid Finely Integrated Product Scanning (HFIPS), and introduce a novel approach for Montgomery multiplication, which we call Hybrid Separated Product Scanning (HSPS). Finally, we show how to perform the modular subtraction of Montgomery reduction in a regular fashion without execution of conditional statements so as to counteract Simple Power Analysis (SPA) attacks. Our AVR implementation of the HFIPS and HSPS method outperforms the Montgomery multiplication of the MIRACL Crypto SDK by up to 21.58% and 14.24%, respectively, and is twice as fast as the modular multiplication of the TinyECC library.

Keywords: AVR architecture, multi-precision arithmetic, hybrid multiplication, modular reduction, SPA countermeasure.

1 Introduction

Long integer modular arithmetic, in particular modular multiplication, is at the heart of many practical public-key cryptosystems, including "traditional" ones that operate in a large ring or group (e.g. RSA [23], DSA [22], Diffie-Hellman [7]), as well as elliptic curve schemes (e.g. ECDSA [22], ECDH [14]) if they use a prime field $\mathbb{F}_p$ as underlying algebraic structure. The major operation of the former class of cryptosystems is exponentiation in either $\mathbb{Z}_n$ or $\mathbb{Z}_p^*$, which can be carried out through modular multiplications and modular squarings [9]. On the other hand, elliptic curve schemes perform scalar multiplication in an additive group, an operation that in turn is composed of additions, multiplications, and inversions in the underlying field [14]. However, most software implementations use projective coordinates to represent points on the curve, thereby trading inversions for multiplications in $\mathbb{F}_p$ to reduce the overall execution time. In this case, the performance of a scalar multiplication is primarily determined by the efficiency of the multiplication in the prime field $\mathbb{F}_p$. Modular multiplication is also a performance-critical arithmetic operation of pairing-based cryptosystems (e.g. identity-based encryption, short signature schemes) [3].

It is common practice in Elliptic Curve Cryptography (ECC) to use primes of a "special" form so as to facilitate the modular reduction [14]. A well-known example are pseudo-Mersenne primes, i.e. primes that are slightly smaller than a power of two and can be written as $p = 2^n - c$ where c is typically chosen to fit into a single register of the target processor. The computational complexity of reduction modulo such primes grows linearly with their length, whereas the reduction operation for general primes has quadratic complexity [14]. A second example of primes that allow one to perform a reduction in linear time are the so-called generalized-Mersenne primes, which are standardized by the National Institute of Standards and Technology (NIST) [22]. Software implementations of ECC often follow a dual approach and support both fast modular reduction techniques for a small set of special primes (e.g. the NIST primes) and a generic reduction routine for "arbitrary" primes. Many cryptographic libraries, such as TinyECC [18] and OpenSSL, take this approach to combine high performance with high flexibility. Therefore, generic modular multiplication techniques, like those introduced by Barrett [4] and Montgomery [21] roughly 30 years ago, are not only important for RSA but also for ECC.

Formally, a modular multiplication $A \cdot B \bmod M$ involves multiplying two n-bit operands A and B, yielding a $2n$-bit product $P = A \cdot B$, followed by the reduction of P modulo M to get a final result in the range of $[0, M-1]$. The latter operation, i.e. the reduction of P with respect to a given modulus M, has a major impact on the execution time of a modular multiplication. A straightforward way to obtain the residue $P \bmod M$ is to divide P by M and find the remainder of this division. However, performing integer division in software is extremely expensive for large operands, which makes this approach unpractical for cryptographic applications. In 1985, Peter Montgomery [21] introduced an efficient (and nowadays widely-used) technique to accomplish a modular reduction without trial division. The basic idea is to replace the modular reduction $P \bmod M$ by a computation of the form $P \cdot 2^{-n} \bmod M$ (where n denotes the bitlength of M), which is much cheaper than computing the actual residue via division. In general, when implemented in software, the Montgomery reduction of a $2n$-bit product P with respect to an n-bit modulus M is just slightly more costly than the multiplication of two n-bit operands [10].

The efficient implementation of multiplication, reduction and other computation-intensive arithmetic operations is particularly challenging for embedded processors with limited resources. The root of the problem is the length of the operands (e.g. 160 bits for an elliptic curve cryptosystem, 1024 bits in the case of RSA), which exceeds the word-size of a small 8 or 16-bit processor by up to two orders of magnitude. Recent research in the area of long-integer arithmetic for such processors focused on the 8-bit AVR architecture [1] (e.g. ATmega128 [2]) as target platform. In 2004, Gura et al published a landmark paper [13] on optimizing modular arithmetic for AVR processors in which they introduce the idea of *hybrid multiplication*. By exploiting the large register file to store (parts of) the operands, the hybrid method allows for a considerable reduction of the number of load instructions compared to a conventional (i.e. column-wise) implementation of multiple-precision multiplication [6, 13]. Gura et al reported an execution time of 3106 clock cycles for a (160×160)-bit multiplication on the ATmega128, a result that was subsequently further improved by Uhsadel et al (2881 cycles [27]), Liu et al (2865 cycles [19]), Zhang et al (2845 cycles [32]), as well as Scott et al (2651 cycles with "unrolled" loops [24]).

In this paper, we continue the line of research described above and advance the state-of-the-art in efficient modular arithmetic for 8-bit AVR processors in three directions. First, we introduce a new variant of the hybrid multiplication technique that is roughly 10% faster than Gura et al's original hybrid method [13]. Our hybrid technique is similar to the one of Zhang et al [32], but benefits from better register allocation and reduced loop overhead (i.e. improved initialization of pointers and more efficient testing of branch conditions). Thanks to our sophisticated register allocation, only 30 (out of 32) AVR working registers are actually occupied during execution of a hybrid multiplication, which allows for easy integration of Montgomery reduction[1]. The second contribution of this paper is a comprehensive performance analysis and comparison of six methods for software implementation of Montgomery multiplication; five are described in [17] and the sixth variant is from [19]. Our results shed some new light on the relative performance of the different Montgomery multiplication methods since they contradict the findings of the current literature, e.g. [17]. Finally, as third contribution, we describe how to perform the final subtraction of M (which is required when a Montgomery product is not fully reduced) in a regular fashion so as to thwart side-channel attacks [20]. Our approach tolerates incompletely-reduced operands and ensures that always the same sequence of instructions is executed, regardless of the actual value of the Montgomery product.

2 Montgomery Modular Multiplication

Montgomery multiplication (named after Peter Montgomery) was originally introduced in 1985 [21] and has since then become one of the most-widely used techniques for high-speed implementation of modular multiplication [8]. In the

[1] The integration of Montgomery reduction into hybrid multiplication (using e.g. the so-called FIOS or FIPS method [17]) can significantly increase the register pressure since two registers are necessary to accommodate the 16-bit pointer to the modulus M. We designed our hybrid multiplication to take this into account by leaving two registers for M, which helps to prevent register spills in the FIPS inner loop.

Algorithm 1. Calculation of the Montgomery product

```
Input: An odd $n$-bit modulus $M$, Montgomery radix $R = 2^n$, two operands $A, B$ in
    the range $[0, M-1]$, and pre-computed constant $M' = -M^{-1} \bmod R$
Output: Montgomery product $Z = \mathrm{MonPro}(A, B) = A \cdot B \cdot R^{-1} \bmod M$
1: $T \leftarrow A \cdot B$
2: $Q \leftarrow T \cdot M' \bmod R$
3: $Z \leftarrow (T + Q \cdot M)/R$
4: if $Z \geq M$ then $Z \leftarrow Z - M$ end if
5: return Z
```

following, we use M to denote an odd modulus consisting of n bits and A, B to denote two residues modulo M, i.e. $0 \leq A, B < M$. Rather than computing the residue of $A \cdot B \bmod M$ directly, Montgomery's algorithm returns the so-called *Montgomery product* of A and B as result, which is defined as follows.

$$\mathrm{MonPro}(A, B) = A \cdot B \cdot R^{-1} \bmod M \tag{1}$$

The factor R in Equation (1) is often referred to as *Montgomery radix* and can be any integer that is bigger than M and relatively prime to it, i.e. R needs to satisfy $\gcd(N, R) = 1$. However, for reasons of implementation efficiency, R is in general a power of two, e.g. $R = 2^n$. The central idea of Montgomery multiplication is to replace the reduction modulo M (which would normally require a costly division by M) by a division by R and a reduction mod R, which are cheap operations when R is a power of two. More precisely, a division by 2^n is merely an n-bit right-shift operation, while a reduction modulo 2^n requires the truncation of all high-order bits above the n-th position. Algorithm 1 specifies the computation of the Montgomery product in detail. In addition to the three operands A, B, and M, the algorithm needs M' as input, which is the inverse of $-M$ (or, more precisely, the inverse of $R - M$) modulo R. However, M' can be pre-computed (using e.g. the Euclidean algorithm as described in [17]) since it depends only on M and R, i.e. M' is fixed for a given M.

Based on Algorithm 1, the Montgomery product $A \cdot B \cdot R^{-1} \bmod M$ can be obtained as follows. First, the n-bit operand A is multiplied by n-bit operand B, giving a $2n$-bit product T. Then, in line 2, the quotient $Q = -\frac{T}{M} \bmod R$ is calculated, which is simply a multiplication of the low-order n bits of T by the pre-computed constant $M' = -M^{-1} \bmod R$ [8]. Note that we actually need to calculate only the lower half (i.e. the n least significant bits) of $T \cdot M'$ because our Montgomery radix R is 2^n. In line 3, a multiplication and a division by R is performed; the latter is just an n-bit right-shift since $R = 2^n$. Thus, we have to calculate only the upper half of the product $Q \cdot M$. The n least significant bits of $T + Q \cdot M$ are 0, which means the division by R (i.e. the n-bit right-shift) in line 3 does not destroy any information. The result Z obtained so far may be not fully reduced (i.e. Z may not be the least non-negative residue modulo M) so that a "final subtraction" of M becomes necessary (line 4). In summary, the computational cost of Algorithm 1 amounts to one conventional multiplication of n-bit operands (line 1) and two "half" multiplications where only either the lower part (line 2) or the upper part (line 3) of the product is really needed. As a consequence, computing the Montgomery product is just slightly more costly than two conventional multiplications.

Software implementations of Algorithm 1 generally store the large integers A, B, and M in arrays of single-precision words (i.e. arrays of `unsigned int` in C and similar programming languages). Assuming a processor with a word-size of w bits, an n-bit integer X consists of $s = \lceil n/w \rceil$ single-precision (i.e. w-bit) words. Throughout this paper, we will use uppercase letters to represent large integers, whereas lowercase letters, usually with a numerical index, will denote individual w-bit words. The most and least significant word of an integer X are x_{s-1} and x_0, respectively, i.e. we have $X = (x_{s-1}, \ldots, x_1, x_0)$. There exist several implementation options and optimization techniques to efficiently perform a Montgomery multiplication in software; they can be categorized according to the order in which the words of the operands (resp. product) are accessed and whether multiplication and modular reduction are carried out *separately* or in an *integrated* fashion (see e.g. [17] for details). In brief, when using the so-called *operand scanning* method, the words of the operands are loaded sequentially, in ascending order, starting with the least significant word. On the other hand, the main characteristic of the *product scanning* technique is that each word of the result is stored (i.e. written to memory) only once, which happens in ascending order [6]. Both methods can be used to implement Montgomery multiplication in either a separated way (i.e. the modular reduction is accomplished after the multiplication) or an integrated way by alternating multiplication and reduction steps. In the latter case, we can further distinguish between a coarse and a fine integration of multiplication and modular reduction. Combinations of all these techniques allow for a multitude of algorithms for calculating the Montgomery product, six of which we briefly describe in the following subsections.

2.1 Separated Operand Scanning (SOS)

In Koç et al's original description of the SOS method, both the multiplication and the reduction are carried out according to the operand-scanning technique [17]. The inner loop of the multiplication (and also that of the reduction) performs operations of the form $(u, v) \leftarrow a \cdot b + c + d$, whereby a, b, c, and d are single-precision integers (i.e. w-bit words) and (u, v) denotes a double-precision (i.e. $2w$-bit) quantity. Each execution of this inner loop on a general-purpose RISC processor, e.g. the ATmega128, involves a `mul` and four `add` (resp. `adc`) instructions[2]. Assuming s-word operands, the operand-scanning multiplication of the SOS method executes s^2 `mul`, $4s^2$ `add` (or `adc`), $2s^2 + s$ `load`, as well as $s^2 + s$ `store` instructions (see Algorithm 1 in [10] for a detailed analysis). The original operand-scanning approach for Montgomery reduction as described in

[2] Note that we count the number of `add` *instructions* (in the same way as [10]), while Koç et al [17] assess the number of add *operations*. Adding a single-precision word to a double-precision quantity (u, v) counts for one add operation, but requires two `add` instructions, one of which is actually an `adc` (add-with-carry).

Algorithm 2. Montgomery reduction (operand scanning form)

Input: An s-word modulus $M = (m_{s-1}, \ldots, m_1, m_0)$, operand $P = (p_{2s-1}, \ldots, p_1, p_0)$ with $P < 2M - 1$, and pre-computed constant $m'_0 = -m_0^{-1} \bmod 2^w$

Output: Montgomery residue $Z = P \cdot 2^{-n} \bmod M$

```
1:  $t \leftarrow 0$
2:  for $i$ from 0 by 1 to $s-1$ do
3:      $u \leftarrow 0$
4:      $q \leftarrow p_i \cdot m'_0 \bmod 2^w$
5:      for $j$ from 0 by 1 to $s-1$ do
6:          $(u, v) \leftarrow m_j \cdot q + p_{i+j} + u$
7:          $p_{i+j} \leftarrow v$
8:      end for
9:      $(u, v) \leftarrow p_{i+s} + u + t$
10:     $p_{i+s} \leftarrow v$
11:     $t \leftarrow u$
12: end for
13: for $j$ from 0 by 1 to $s-1$ do
14:     $z_j \leftarrow p_{j+s}$
15: end for
16: $z_s \leftarrow t$
17: if $Z \geq M$ then $Z \leftarrow Z - M$ end if
```

Section 4 of [17] employs a special ADD function to propagate a carry bit up to the most significant word. Our implementation simply holds the carry bit in an extra register t and adds it in the next iteration of the outer loop as shown in Algorithm 2. In this way, the operand-scanning form of Montgomery reduction consists of $s^2 + s$ `mul`, $4s^2 + 2s$ `add` or `adc`, $2s^2 + 2s + 1$ `load`, and $s^2 + 2s + 1$ `store` instructions, which means the SOS method (excluding final subtraction) needs to execute $2s^2 + s$ `mul`, $8s^2 + 2s$ `add` (resp. `adc`), $4s^2 + 3s + 1$ `load`, and $2s^2 + 3s + 1$ `store` instructions altogether.

2.2 Finely Integrated Product Scanning (FIPS)

The FIPS method (Algorithm 1 in [11]), originally introduced in [8], performs multiplication and reduction steps in a "finely" interleaved fashion in the same inner loop. From an algorithmic viewpoint, the FIPS technique consists of two nested loops; both inner loops compute (parts of) the product $A \cdot B$ and then add (parts of) the product $Q \cdot M$ to it. After the first inner loop, a word of the quotient Q is calculated with help of the least-significant word of M' (i.e. the pre-computed constant $m'_0 = -m_0^{-1} \bmod 2^w$ [17]) and temporarily stored in the array of the final result. The least-significant word of the intermediate sum obtained at the end of the second inner loop is always zero, which means it can be right-shifted by w bits without "destroying" any information. In each iteration of the second outer loop, a word of the result (i.e. the Montgomery product) is obtained and written to memory. Note that this result consists of $s + 1$ words (whereby the MSW is either 0 or 1) since it may be incompletely reduced.

In each iteration of one of the inner loops, two multiply-accumulate (MAC) operations of the form $(t, u, v) \leftarrow (t, u, v) + a \cdot b$ are carried out, i.e. two words are multiplied and the double-precision product is added to a cumulative sum held in the three registers v, u and t. Note that Koç et al [17] employ a special `ADD` function to process carries (similar to the SOS method), but we avoid this by using three registers to hold the cumulative sum. The inner-loop operation of our FIPS method is identical to that of the product-scanning multiplication [14] and needs one `mul` and three `add` instructions. In total, the FIPS method requires $2s^2 + s$ `mul`, $6s^2$ `add/adc`, $4s^2 - s$ `load`, and $2s + 1$ `store` instructions altogether (excluding final subtraction) [10].

2.3 Coarsely Integrated Operand Scanning (CIOS)

Instead of computing the complete multiplication first and doing the reduction afterwards (like in Section 2.1), the CIOS method performs multiplication and reduction in an interleaved fashion, similar to Section 2.2. Algorithm 4 in [10] describes the CIOS method in detail; it consists of an outer loop that contains two inner loops. The first inner loop calculates parts of the product $A \cdot B$ and stores the intermediate result in an array in RAM. After the first inner loop, a word of the quotient Q is determined, which is subsequently used in the second inner loop to get a multiple of M to be added to the intermediate result. This addition zeroes out the least significant word of the intermediate result and so contributes to the modular reduction. A w-bit right-shift operation is implicitly performed in the second inner loop through indexing, i.e. by writing a word with index i to the $(i-1)$-th position in the target array. The two inner loops execute exactly the same operation as the SOS method, namely a computation of the form $(u, v) \leftarrow a \cdot b + c + d$. We eventually obtain a result that consists of $s + 1$ words (with the most-significant word being either 0 or 1), which means a final subtraction of M may be necessary to get a fully reduced result [17]. In total, the CIOS method requires $2s^2 + s$ `mul`, $8s^2 + 4s$ `add`, $4s^2 + 5s$ `load`, and $2s^2 + 3s$ `store` instructions (see [10] for further details[3]).

2.4 Coarsely Integrated Hybrid Scanning (CIHS)

This method, introduced in [17, Section 8], is related to both the SOS and the CIOS approach sketched before. It is called "hybrid scanning" method because it mixes operand scanning and product scanning for multiplication, while the reduction operation is accomplished solely in operand-scanning form. The CIHS method consists of two outer loops and three inner loops. The first outer loop computes a part of the product $A \cdot B$, while the second outer loop contributes to the reduction operation and the rest of the multiplication. Furthermore, the second outer loop shifts the intermediate result one word (i.e. by w bits) to the right in each iteration. The "splitting" of the multiplication is possible since, in

[3] Note that the number of `add` (resp. `adc`) instructions for the CIOS method specified in Table 4 of [10] is wrong; the correct number is $8s^2 + 4s$ for s-word operands.

the course of Montgomery modular reduction, the variable m computed at the beginning of the second outer loop only depends on t_0. The operation executed by the first two inner loops is exactly the same as that of the SOS and CIOS method, respectively. However, the third inner loop is slightly simpler because it performs an operation of the form $(u, v) \leftarrow a \cdot b + c$, each execution of which costs one `mul` and two `add` (resp. `adc`) instructions. Putting it all together, the CIHS method requires $2s^2 + s$ `mul`, $9s^2 + 5s$ `add`/`adc`, $11s^2/2 + 7s/2$ `load`, as well as $3s^2 + 2s$ `store` instructions (excluding the final subtraction).

2.5 Finely Integrated Operand Scanning (FIOS)

The last operand-scanning variant of Montgomery multiplication we discuss in this paper is the Finely Integrated Operand Scanning (FIOS) method, given in [12, Algorithm 1]. Compared to the four methods outlined before, the structure of this algorithm is very simple as it comprises just an outer loop with a single inner loop. The inner loop of the FIOS variant described in [12] executes two operations of the form $(u, v) \leftarrow a \cdot b + c + d$, one contributes to the calculation of the product of A and B, and the other to the Montgomery reduction of this product. Similar to the CIOS method, the quality of the implementation of the inner-loop operation has a major impact on the algorithm's overall execution time. In summary, the FIOS method of Montgomery multiplication requires to perform $2s^2 + s$ `mul`, $8s^2$ `add`, $3s^2 + 4s$ `load`, and $s^2 + s$ `store` instructions.

2.6 Separated Product Scanning (SPS)

The Montgomery multiplication methods sketched in the previous five subsections were first described and analyzed by Koç et al [17]. In this subsection, we present a sixth method, which we call *Separated Product Scanning (SPS)*. The SPS method separates multiplication steps and reduction steps (similar to the SOS method), i.e. the Montgomery reduction is carried out as a self-contained operation *after* the multiplication. As its name suggests, the SPS technique is based on the product scanning approach for multiplication (see Algorithm 2 in [10]) and then uses the product-scanning form of Montgomery reduction shown in Algorithm 3. More details on this product-scanning based Montgomery reduction can be found in [10, 19]. The SPS method was originally introduced in [19] as a product-scanning variant of the SOS technique, but we feel that the name "Separated Product Scanning" better denotes the characteristics of this method. As per [10], a product-scanning multiplication of two s-word operands consists of s^2 `mul`, $3s^2$ `add`, $2s^2$ `load`, and $2s$ `store` instructions. Algorithm 3 requires $s^2 + s$ `mul`, $3s^2 + 6s$ `add`, $2s^2 + 2s$ `load`, and $2s + 1$ `store` instructions [10], which amounts to $2s^2 + s$ `mul`, $6s^2 + 6s$ `add` (or `adc`), $4s^2 + 2s$ `load`, and $4s + 1$ `store` instructions for the complete SPS method.

2.7 Analysis and Comparison

Table 1 summarizes and compares the base instruction counts of all six Montgomery multiplication techniques considered in this section. The two variants

Algorithm 3. Montgomery reduction (product scanning form) [10, Algorithm 5]

Input: An s-word modulus $M = (m_{s-1}, \ldots, m_1, m_0)$, a product P in the range of $[0, 2M-2]$, pre-computed constant $m'_0 = -m_0^{-1} \bmod 2^w$
Output: Montgomery residue $Z = P \cdot 2^{-n} \bmod M$

```
1: $(t, u, v) \leftarrow 0$
2: for $i$ from 0 by 1 to $s-1$ do
3:     for $j$ from 0 by 1 to $i-1$ do
4:         $(t, u, v) \leftarrow (t, u, v) + z_j \cdot m_{i-j}$
5:     end for
6:     $(t, u, v) \leftarrow (t, u, v) + p_i$
7:     $z_i \leftarrow v \cdot m'_0 \bmod 2^w$
8:     $(t, u, v) \leftarrow (t, u, v) + z_i \cdot m_0$
9:     $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
10: end for
11: for $i$ from $s$ by 1 to $2s-2$ do
12:     for $j$ from $i-s+1$ by 1 to $s-1$ do
13:         $(t, u, v) \leftarrow (t, u, v) + z_j \cdot m_{i-j}$
14:     end for
15:     $(t, u, v) \leftarrow (t, u, v) + p_i$
16:     $z_{i-s} \leftarrow v$
17:     $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
18: end for
19: $(t, u, v) \leftarrow (t, u, v) + p_{2s-1}$
20: $z_{s-1} \leftarrow v,\ z_s \leftarrow u$
21: if $Z \geq M$ then $Z \leftarrow Z - M$ end if
```

Table 1. Comparison of base instructions for Multiplication modular multiplications (excluding final subtraction)

Algorithm	# `mul`	# `add`	# `load`	# `store`
FIPS	$2s^2 + s$	$6s^2$	$4s^2 - s$	$2s + 1$
SPS	$2s^2 + s$	$6s^2 + 6s$	$4s^2 + 2s$	$4s + 1$
CIOS	$2s^2 + s$	$8s^2 + 4s$	$4s^2 + 5s$	$2s^2 + 3s$
SOS	$2s^2 + s$	$8s^2 + 2s$	$4s^2 + 3s + 1$	$2s^2 + 3s + 1$
CIHS	$2s^2 + s$	$9s^2 + 5s$	$11s^2/2 + 7s/2$	$3s^2 + 2s$
FIOS	$2s^2 + s$	$8s^2$	$3s^2 + 4s$	$s^2 + s$

based on the product-scanning method (i.e. FIPS and SPS) execute multiply-accumulate operations of the form $(t, u, v) \leftarrow (t, u, v) + a \cdot b$ in the inner loops [10], whereby each operation involves three `add` or `adc` instructions to add the product $a \cdot b$ to a cumulative sum. Consequently, the FIPS and SPS technique execute three `add` (resp. `adc`) per one `mul` instruction. On the other hand, the operand-scanning variants feature a common inner-loop operation of the form $(u, v) \leftarrow a \cdot b + c + d$, which costs four `add`/`adc` per `mul` instruction. A second major difference between the product-scanning variants and their counterparts based on the operand-scanning technique is the number of `store` instructions as shown in the last column of Table 1. The former execute `store` instructions solely in the outer loops, whereas the latter perform stores in the inner loop(s) [10]. Therefore, the number of `store` instructions carried out by FIPS and SPS increases linearly with the number of words. The operand-scanning variants, on the other hand, exhibit a quadratic growth of the number of stores.

Our analysis of the base instructions indicates a clear advantage of the two product-scanning methods, which will be confirmed by implementation results in Section 4. However, our analysis is not in agreement with that of Koç et al [17], who clearly identified the CIOS method as the most efficient one on basis of both their theoretical cost model and measured results. As stated in Section 2.1, this deviation can be explained by differences in the underlying cost model since Koç et al consider the number of basic operations, whereas we count the number of basic instructions as this is more accurate. Furthermore, Koç et al use a special `ADD` function to propagate carries in their SOS, FIOS, and FIPS method, which we do not need since we hold all carries in registers.

3 Our Implementation

In this section, we first introduce a novel variant of the hybrid multiplication method, which saves 10.6% execution time compared to the original one from [13]. Then, we combine our hybrid multiplication with Montgomery's algorithm to obtain different variants of a hybrid Montgomery multiplication. Finally, we describe an efficient implementation of the conditional subtraction of M.

3.1 Optimized Hybrid Multiplication

A straightforward implementation of the product-scanning method processes a single word of operand A and operand B at a time; therefore, in each iteration of the inner loop, a word of each A and B is loaded from RAM, multiplied, and added to a cumulative sum [6]. Gura et al [13] observed that the performance of the product-scanning method can be significantly improved if several words of the operands are processed in each iteration. This approach is, in essence, a special form of loop unrolling and particularly efficient on processors featuring a large number of registers. Taking the 8-bit AVR platform [1] as example, we can easily process $d = 4$ (or even $d = 5$) bytes of the operands at a time, and so reduce the number of loop iterations by a factor of d. In each iteration of the inner loop, four bytes (i.e. 32 bits) of A and B are loaded from memory and multiplied together to yield a 8-byte (i.e. 64-bit) result, which is then added to a cumulative sum held in nine registers. Gura et al used the operand-scanning approach for the 4-byte-by-4-byte (i.e. $(32 \times 32\text{-bit})$-bit) multiplications in the inner loop as illustrated on the left of Figure 1. This multiplication technique is referred to as "hybrid multiplication" because it combines product scanning in the outer loop with operand scanning in the inner loop(s). The main advantage of hybrid multiplication is a reduced number of load instructions compared to the straightforward product-scanning method (see [13] for details).

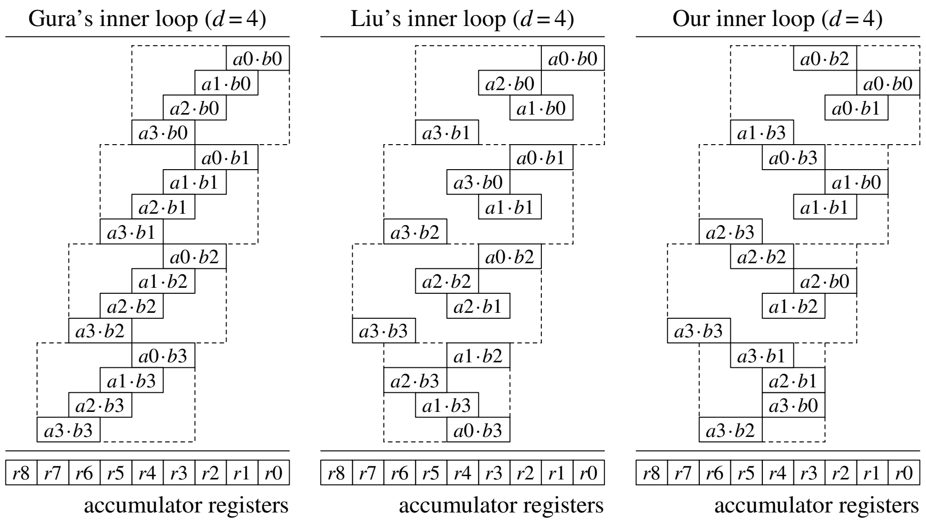

Fig. 1. Comparison of inner-loop operation for hybrid multiplication

In recent years, there have been several attempts to improve the inner-loop operation of the hybrid method, taking the properties of the AVR architecture into account[4]. For example, Liu et al re-arranged in [19] the order of the multiplications in the inner loop (depicted in the middle of Figure 1), which allowed them to decrease the number of `mov` (resp. `movw`) instructions compared to the original hybrid method. Scott et al [24] used so-called "carry catcher" registers to limit the propagation of carries and totally unrolled the loops to achieve an extra speed-up. Our implementation of the inner loop, shown on the right side of Figure 1, is inspired by both Liu et al and Scott et al. Just like Liu et al, we schedule the `mul` instructions in a special order with the goal of reducing the computational cost of the inner loop. If we assume $d = 4$, the 16 byte products are calculated as shown in Figure 1, whereby the execution time elapses from top to bottom, i.e. $a_0 \cdot b_2$ is the first byte product we generate and $a_3 \cdot b_2$ the last. Our variant of the inner-loop operation borrows the idea of catching carry bits from [24], but we do not use separate registers for that purpose.

To simplify the explanation of our inner loop, we split the 16 byte-products into four blocks, indicated by dashed boxes in Figure 1. At the beginning, four bytes of operand B (labeled b_0, b_1, b_2 and b_3 in Figure 1) along with two bytes of A (namely a_0 and a_1) are loaded from RAM. We first multiply a_0 by b_2 and copy the 16-bit product to two temporary registers, t_0 and t_1, with help of the `movw` instruction. The register t_1 holds the "upper" (i.e. more significant) byte of the product and t_0 the "lower" byte. Next, we form the product $a_0 \cdot b_0$ and add it along with the content of t_0 to the three accumulator registers r_0, r_1 and

[4] A special "feature" of AVR is that the `mul` instruction modifies the carry flag, which complicates the implementation of multi-precision multiplication.

r_2. A potential carry from this addition can be safely added into the temporary register t_1 without overflowing it since the upper byte of the product of two 8-bit integers is always smaller than 255. Thereafter, we multiply a_0 by b_1, add the resulting 16-bit product $a_0 \cdot b_1$ to r_1, r_2, and propagate the carry from the last addition to the temporary register t_1. Again, it is not possible to overflow t_1, not even in the most extreme case where the operand bytes a_0, b_0, b_1, and b_2 as well as the involved accumulator bytes r_0, r_1, and r_2 have the maximum value of 255. After computation of the last product of the first block (which is $a_1 \cdot b_3$), we add t_1 and $a_1 \cdot b_3$ to the three accumulator registers r_3, r_4, r_5, and finally propagate the carry bit from the last addition up to r_8. In summary, the processing of the first block in Figure 1 requires four `mul`, a `movw`, and a total of 13 `add` or `adc` instructions, respectively.

The next two blocks are processed in essentially the same way as the first block; the only actual difference is the loading of the remaining operand bytes of A, namely a_2 and a_3, which is done during the second and third block, respectively. Again, we use temporary register t_1 to catch the carries generated in the addition of the second and third byte-product of the respective block. The loading of operand byte a_2 is part of the second block and performed after the multiplication of a_0 by b_3. Note that the byte a_0 is not needed anymore once $a_0 \cdot b_3$ has been produced, which means we can load a_2 into the register holding a_0. The operand byte a_3 gets loaded after the multiplication of a_1 by b_2 in the third block. At that time, the byte a_1 is not needed anymore, and hence we can load a_3 into the same register, thereby overwriting a_1. In summary, the second and third block execute 12 and 11 `add` (or `adc`) instructions, respectively. The number of `mul` and `movw` instructions are the same as for the first block.

The fourth block, in which the remaining four byte-products are generated and added to the accumulator registers, differs a bit from the former three. We first multiply a_3 by b_1 and move the resulting 16-bit product to the temporary register pair t_1, t_0. Then, we compute the product $a_2 \cdot b_1$, add its lower byte to the accumulator register r_3 and the upper byte to the two temporary registers holding $a_3 \cdot b_1$. The last addition does not produce a "carry out," which means this addition can not overflow the temporary register pair. Next in schedule is the third product $a_3 \cdot b_0$; it is processed in the same way as before and can also not overflow the registers t_1, t_0. After finally multiplying a_3 by b_2, the temporary register t_0 is added to r_4, and a possible carry bit is added with t_1 to the product $a_2 \cdot b_3$. The obtained sum is then added to the accumulator registers r_5, r_6 and the carry from the last addition is propagated to r_8. All in all, the fourth block requires to execute 13 `add` (resp. `adc`) instructions, very similar to the first block. The complete inner-loop operation for $d = 4$ consists of a total of 46 `add` (or `adc`), 16 `mul`, eight `ld` (i.e. load), and four `movw` instructions. On an ATmega128 processor [2], these instruction counts translate to an execution time of 101 clock cycles per iteration of the inner loop (including update of the loop-control variable and branch instruction). Another property of our loop is its economic register usage; it occupies only 30 out of the 32 available registers [1], which simplifies the implementation of Montgomery multiplication.

Table 2. Comparison of instruction counts for 160-bit multi-precision multiplication on the ATmega128 (without function call overhead)

Instruction type	`add`	`mul`	`ld`	`st`	`mov`	Other	Total
CPI	1	2	2	2	1	cycles	cycles
Classic Comba	1200	400	800	40	81	44	3805
Gura et al [13]	1360	400	167	40	355	197	3106
Uhsadel et al [27]	986	400	238	40	355	184	2881
Liu et al [19]	1194	400	200	40	212	179	2865
Zhang et al [32]	1092	400	200	20	202	271	2845
Our work (parameterised)	1213	400	200	40	100	185	**2778**
Hutter et al [15] (looped)	1252	400	92	66	41	276	2685
Scott et al [24] (unrolled)	1263	400	200	40	70	38	2651
Hutter et al [15] (unrolled)	1240	400	80	60	2	68	2395
Seo et al [25] (unrolled)	1240	400	70	60	n/a	56	2356
Seo et al [26] (unrolled)	1230	400	70	60	n/a	56	2346

3.2 Evaluation of our Optimized Hybrid Multiplication

Table 2 shows the instruction counts and total execution time (in clock cycles) of our improved hybrid method for a (160×160)-bit multiplication on an ATmega128 processor [2]. We use (160×160)-bit multiplication as benchmark to allow for a direct comparison with past work that targeted ECC. Note that the instruction numbers in the columns labeled with `add`, `ld`, and `mov` also include `adc`, `ldd`, and `movw`, respectively (i.e. we do not differentiate between `add` and `adc` as they both require a single cycle on AVR processors). Our variant of the hybrid method executes a (160×160)-bit multiplication in just 2778 cycles on the ATmega128, which is approximately 10.6% faster than the original hybrid method of Gura et al [13]. This saving in execution time is mainly due to the fact that we have to carry out only 100 `mov` (resp. `movw`) instructions, whereas Gura et al need 355 `mov` or `movw` instructions. Furthermore, our special scheduling of the multiplications in the inner loops reduces the number of `add` (and `adc`) instructions, similar to the implementations described in [19] or [32]. The hybrid multiplication technique of Uhsadel et al [27] requires 2881 cycles, even though their implementation (as well as the one of Gura et al [13]) is based on $d = 5$ for 160-bit operands instead of $d = 4$ as in our work.

In general, when analyzing different software libraries for multiple-precision arithmetic, one has to distinguish three implementation options with respect to the processing of loops: unrolled, looped, and parameterized. Loop unrolling is well known to improve performance as it eliminates the loop overhead (such as the updating of a loop counter or execution of a branch instruction) and allows for some extra optimizations. For example, the first and last iteration of a loop often differs from the middle iterations and can, therefore, be specifically tuned when the loop is unrolled. The drawbacks of loop unrolling are large code size (i.e. increased program memory) and poor flexibility (resp. scalability) since an

Table 3. Comparison of code size (in bytes) of "conventional" multiplication (without reduction) for operand lengths ranging from 160 to 1024 bits

Implementation	160	192	224	256	512	1024
Hutter et al [15] (looped)	1562	1866	1538	1766	1544	1572
Hutter et al [15] (unrolled)	3778	5436	7340	9558	37884	151044
Our work (parameterised)	514	514	514	514	514	514

unrolled implementation supports just a single operand length. At the opposite end of the design space are parameterized implementations, which allow one to pass the operand length as a parameter to a function call. Such parameterized implementations are very flexible since one and the same function can process operands of any size, but this flexibility comes at the expense of decreased performance due to the fact that (full) loop unrolling and other optimizations are not possible anymore. Somewhere in the middle between these two approaches are looped implementations, which have "rolled" loops but still support only a single operand length. Looped implementations outperform their parameterized counterparts since they provide more avenues for optimization. Having a fixed operand helps to improve the performance as the number of loop iterations is constant and can therefore be "hard-coded." Thus, it is not necessary to waste a register for storing the operand length, which leaves more registers available for the actual computation.

Even though our implementation of the hybrid method is parameterized, it compares very well with looped and unrolled implementations. For example, the looped version of Hutter et al's operand caching technique [15] is just 93 cycles faster than our work (2685 vs. 2778 cycles, see Table 2), even though their code is optimized for 160-bit operands, while our implementation supports operands of any length. However, this slight performance gain comes at the cost of three times larger codes size, which can be seen from Table 3. Furthermore, one has to consider that Hutter et al achieved their execution time of 2685 clock cycles by using all 32 available registers[5] of the ATmega128. The unrolled implementations from [15, 24–26], while being fast, suffer from a prohibitively large code size, especially for operands exceeding 256 bits in size (see Table 3). Full loop unrolling may be a viable optimization for ECC, but not for RSA.

3.3 Hybrid Montgomery Multiplication

Similar to the "ordinary" multiplication (without modular reduction), also the six Montgomery multiplication techniques described in this paper can be made

[5] Note that the fastest implementation of a conventional multiplication (i.e. a multiplication without reduction) does not necessarily lead to the fastest implementation of Montgomery multiplication. Generic algorithms for modular multiplication have three input operands (namely A, B, and M), which increases the register pressure compared to an ordinary multiplication. Our variant of the hybrid method occupies only 30 registers and, thus, allows for easy integration of Montgomery reduction.

significantly faster by applying the hybrid method in order to take advantage of the large register file of the AVR platform [1]. Processing several bytes of the operands in each inner-loop iteration yields a performance gain by reducing the number of loads/stores and loop overhead. By combining the hybrid technique with the six Montgomery variants, we get six hybrid Montgomery multiplication methods, which we call hybrid SOS (HSOS), hybrid FIPS (HFIPS), hybrid CIOS (HCIOS), hybrid CIHS (HCIHS), hybrid FIOS (HFIOS), and hybrid SPS (HSPS). Our implementations of these six algorithms have in common that, in each iteration of the inner loop, four bytes of the operands are loaded into the register file and the total number of loop iterations is accordingly reduced by a factor of four compared to the corresponding straightforward (i.e. non-hybrid) Montgomery multiplication technique.

The hybrid product-scanning techniques, namely HFIPS and HSPS, execute operations of the form $(t, u, v) \leftarrow (t, u, v) + a \cdot b$ in the inner loops, whereby the two operand words a and b consist of four bytes each. A total of nine registers is necessary to hold the cumulative sum (t, u, v). Therefore, we can employ the highly-optimized hybrid implementation of the inner-loop operation shown on the right of Figure 1 and explained in detail in Section 3.1. Unlike HSPS, the HFIPS method has to keep four pointers (namely the pointers to the arrays in which the two operands A, B, the result Z, and the modulus M are stored) in registers during the execution of the inner loop to reach top performance. The inner-loop implementation from Subsection 3.1 is ideally suited for the HFIPS method since it needs only 30 registers so that the remaining two registers can be used to hold the pointer to M. The four hybrid Montgomery multiplication methods based on operand-scanning (i.e. HSOS, HCIOS, HCIHS, and HFIOS) have a slightly different inner loop due to the fact that they execute operations of the form $(u, v) \leftarrow a \cdot b + c + d$ and $(u, v) \leftarrow a \cdot b + c$. We implemented these operations to process four bytes at once (i.e. per loop iteration) and optimized them following exactly the same strategies as discussed in Section 3.1.

3.4 Regular Execution of Final Subtraction

As shown in Algorithm 1, the calculation of the Montgomery product may require a final subtraction of the modulus M to get a fully reduced result in the range of $[0, M - 1]$. However, this final subtraction is not carried out when the intermediate result after step 3 of Algorithm 1 is already smaller than M. It is well known that such a conditional execution of a subtraction typically entails observable differences in the power consumption profile, which can be exploited to mount an SPA attack as described in [30] for RSA and in [29] for an elliptic curve cryptosystem. Walter proposed in [28] a smart approach to eliminate the final subtraction by using a larger Montgomery radix of e.g. $R = 2^{n+2}$ instead of $R = 2^n$ and adapting the Montgomery algorithm accordingly. However, this approach requires to calculate the Montgomery product with longer operands (since, as in our case, the operand length must be a multiple of 32), which can severely degrade performance. To overcome this problem, we implemented the final subtraction in an unconditional way by "zeroing out" the words m_i of the

Algorithm 4. Final subtraction without conditional statements

```
Input: $(s+1)$-word Montgomery product $Z = (z_s, z_{s-1}, \ldots, z_1, z_0)$ with $z_s \in \{0,1\}$
    and $s$-word modulus $M = (m_{s-1}, \ldots, m_1, m_0)$
Output: $Z = Z - M$ if $z_s = 1$, otherwise, $Z = Z - 0$
1: $mask \leftarrow -z_s \bmod 2^w$    $\{w$ is the bitlength of a word$\}$
2: $(\varepsilon, z_0) \leftarrow z_0 - (m_i \;\&\; mask)$
3: for $i$ from 1 by 1 to $s-1$ do
4:     $(\varepsilon, z_i) \leftarrow z_i - (m_i \;\&\; mask) - \varepsilon$
5: end for
6: return $Z = (z_{s-1}, \ldots, z_1, z_0)$
```

modulus M, if necessary, as shown in Algorithm 4. The notation in Algorithm 4 follows that of [14], i.e. the word-subtractions are carried out with help of an "subtract with borrow" instruction whereby ε represents the borrow bit.

Based on the concept of incomplete modular arithmetic [31], we do not perform an exact comparison between Z and M, but rather use the value of the most significant word z_s of Z to determine whether Z is too big or not. More precisely, we use z_s to derive a mask that is either a zero word (if $z_s = 0$) or an "all 1" word (if $z_s = 1$). As shown in line 1 of Algorithm 4, such a mask can be simply generated by forming the two's complement of z_s. The mask is applied to the bytes of M (i.e. each m_i is logically ANDed with the mask) before they are subtracted from the words z_i using subtract-with-borrow instructions. In this way, we either subtract the modulus M from product Z (if $z_s = 1$) or we subtract 0 (if $z_s = 0$) so that Z remains the same. The final result may not be the least non-negative residue, but is always in the range $[0, 2^n - 1]$ and hence fits into s words. This incomplete reduction does not introduce any problems in practice since the n-bit result, even if not fully reduced, can still be used as operand in a subsequent Montgomery multiplication (see [31] for details).

4 Performance Evaluation and Comparison

We implemented the six hybrid Montgomery multiplication algorithms in AVR assembly language and evaluated their performance for operands ranging from 160 to 1024 bits. Table 4 shows the simulated execution times we obtained on an ATmega128 processor [2]; these figures include time for the unconditional final subtraction introduced in Section 3.4. Our fastest method, HFIPS, only needs 6080 clock cycles to perform a full 160-bit Montgomery multiplication, which is approximately 1.4 times faster than the slowest algorithm, namely HFIOS. All obtained execution times are visualized on the left of Figure 2.

Besides the computational complexity of algorithms themselves, there are a few other factors affecting the actual performance of the various multiplication methods. For example, the overhead for controlling the loop or the cost to find the correct start address of arrays also impact the execution time. Our results indicate that the interleaved versions of hybrid Montgomery multiplication are sightly faster than the separated versions, e.g. HFIPS outperforms HSPS, and

Table 4. Execution time (in clock cycles) of six hybrid Montgomery multiplication techniques for different operand lengths

Algorithm	160	192	224	256	512	768	1024
HFIPS	6080	8539	11420	14723	56339	124964	220596
HSPS	6648	9171	12110	15465	57281	125722	221044
HCIOS	7140	9983	13310	17121	65033	143922	253787
HSOS	7921	10956	14500	18553	69301	152626	268788
HCIHS	8127	11385	15197	19563	74435	164764	290549
HFIOS	8216	11660	15716	20384	79760	178315	316018

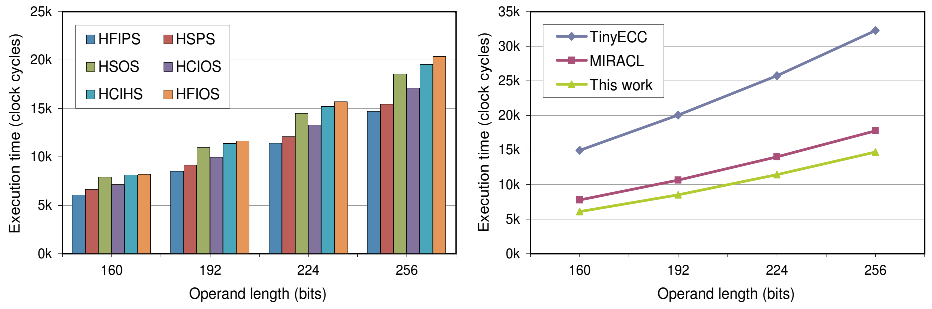

Fig. 2. Performance comparison of our six Montgomery algorithms (left) and comparison or our HFIPS method with Miracl and TinyECC (right)

HCIOS is faster than HSOS. This is mainly because the interleaved versions, in general, incur less overhead than the separated versions (i.e. reduced overhead for controlling loops, handling pointers, and calculating start addresses).

The HCIHS and HFIOS method are the slowest of the six hybrid Montgomery multiplication techniques shown in Table 4. The poor performance of the HCIHS approach is primarily due to the overhead caused by frequent loadings of operands into registers. On the other hand, HFIOS uses a lot of time for the pointer arithmetic required to obtain the correct start address of the operands at the beginning of a loop. Another disadvantage of this method is that it has to handle six variables, namely a_j, b_i, m_j, q, t, and z_j, in the inner loop. Since the hybrid multiplication of $a_j \cdot b_i$ occupies almost all of the 32 working registers, a number of expensive `push` and `pop` operations are required to save pointers on the stack. The cost of the stack operations in HFIOS is higher than cost of the frequent operand loadings in HCIHS; thus, HFIOS is slower than HCIHS.

Table 5 compares our hybrid product-scanning methods, namely HSPS and HFIPS, with the two popular cryptographic libraries TinyECC [18] and Miracl [5] for operands ranging from 160 to 1024 bits in size. The right side of Figure 2 visualizes the execution times of TinyECC, Miracl, and HFIPS, which is the fastest of our six implementations of Montgomery multiplication. To ensure a fair comparison, we downloaded the source code of TinyECC and Miracl from

Table 5. Montgomery Multiplication timings (in clock cycles) of TinyECC, Miracl, and our implementation of the HSPS and HFIPS method

Implementation	160	192	224	256	512	1024
TinyECC [18]	14929	20060	25765	n/a	n/a	n/a
Miracl [5]	7753	10653	14033	17761	58806	221329
This work (HSPS)	6648	9171	12110	15465	57281	221044
This work (HFIPS)	6080	8539	11420	14723	56339	220596

the corresponding home pages, compiled them with AVR studio, and simulated the execution times in a coherent fashion. Both our HFIPS and HSPS method are more than twice as fast as the modular multiplication of TinyECC. On the other hand, compared to the Montgomery multiplication of Miracl, our HFIPS method saves 21.6%, 19.8%, 18.6%, 17.1% execution time for 160, 192, 224, and 256-bit operands, respectively. Note that the performance gap between HFIPS and Miracl becomes smaller when the operand size grows above 256 bits since Miracl employs the asymptotically faster Karatsuba technique [16] to speed up multiplication when the operand length exceeds a certain threshold.

5 Conclusions

The contribution of this work is threefold. First, we presented a new approach to implement hybrid multiplication, saving 10.6% execution time compared to the original method of Gura et al (CHES 2004). This performance gain is achieved by re-ordering the sequence of multiplications in the inner loop along with an efficient way of catching carries, thereby reducing the total number of `add` and `mov` (resp. `movw`) instructions. Another advantage of our hybrid technique is its suitability to implement interleaved variants Montgomery multiplication since it occupies only 30 registers of an AVR processor. Our second contribution is a through analysis and comparison of six hybrid variants of Montgomery modular multiplication. Based on a more precise cost model along with some small optimizations (e.g. elimination of the `ADD` function for carry propagation), we conclude that the FIPS and SPS method reach the best performance, which is contradicting previous results of Koç et al, who found the CIOS method to be superior. A detailed benchmarking on an 8-bit ATmega128 processor confirms our theoretical evaluation and shows that the hybrid FIPS technique requires merely 6080 clock cycles to execute a 160-bit Montgomery multiplication. This result sets a new speed record for modular multiplication on an 8-bit platform and outperforms the Miracl library by more than 20%. Our implementation is parameterized and very compact in terms of code size. The third contribution of this paper is a simple yet efficient approach to perform the conditional final subtraction in an unconditional way by "zeroing out" the words of the modulus if the intermediate result is already smaller than 2^n. This ensures that always exactly the same sequence of instructions is executed, regardless of the actual value of the operands, which helps to thwart certain side-channel attacks.

References

1. Atmel Corporation. 8-bit ARV® Instruction Set. User Guide, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, July 2008.
2. Atmel Corporation. 8-bit ARV® Microcontroller with 128K Bytes In-System Programmable Flash: ATmega128, ATmega128L. Datasheet, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, June 2008.
3. P. S. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In M. Yung, editor, *Advances in Cryptology — CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 354–368. Springer Verlag, 2002.
4. P. D. Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer Verlag, 1987.
5. CertiVox Corporation. CertiVox MIRACL SDK. Source code, available for download at `http://www.certivox.com`, June 2012.
6. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Dec. 1990.
7. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov. 1976.
8. S. R. Dussé and B. S. Kaliski. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 230–244. Springer Verlag, 1991.
9. D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, Apr. 1998.
10. J. Großschädl, R. M. Avanzi, E. Savaş, 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 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 75–90. Springer Verlag, 2005.
11. J. Großschädl and G.-A. Kamendje. Architectural enhancements for Montgomery multiplication on embedded RISC processors. In J. Zhou, M. Yung, and Y. Han, editors, *Applied Cryptography and Network Security — ACNS 2003*, volume 2846 of *Lecture Notes in Computer Science*, pages 418–434. Springer Verlag, 2003.
12. J. Großschädl and G.-A. Kamendje. Optimized RISC architecture for multiple-precision modular arithmetic. In D. Hutter, G. Müller, W. Stephan, and M. Ullmann, editors, *Security in Pervasive Computing — SPC 2003*, volume 2802 of *Lecture Notes in Computer Science*, pages 253–270. Springer Verlag, 2003.
13. N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Verlag, 2004.
14. D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer Verlag, 2004.
15. M. Hutter and E. Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems — CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer Verlag, 2011.

16. A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics - Doklady*, 7(7):595–596, Jan. 1963.
17. Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
18. A. Liu and P. Ning. TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN 2008)*, pages 245–256. IEEE Computer Society Press, 2008.
19. Z. Liu, J. Großschädl, and I. Kizhvatov. Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers. In *Proceedings of the 1st International Workshop on the Security of the Internet of Things (SECIOT 2010)*.
20. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Verlag, 2007.
21. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
22. National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). FIPS Publication 186-4, available for download at `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`, July 2013.
23. R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.
24. M. Scott and P. Szczechowiak. Optimizing multiprecision multiplication for public key cryptography. Cryptology ePrint Archive, Report 2007/299, 2007. Available for download at `http://eprint.iacr.org`.
25. H. Seo and H. Kim. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In D. H. Lee and M. Yung, editors, *Information Security Applications — WISA 2012*, volume 7690 of *Lecture Notes in Computer Science*, pages 55–67. Springer Verlag, 2012.
26. H. Seo and H. Kim. Optimized multi-precision multiplication for public-key cryptography on embedded microprocessors. *International Journal of Computer and Communication Engineering*, 2(3):255–259, May 2013.
27. L. Uhsadel, A. Poschmann, and C. Paar. Enabling full-size public-key algorithms on 8-bit sensor nodes. In F. Stajano, C. Meadows, S. Capkun, and T. Moore, editors, *Security and Privacy in Ad-hoc and Sensor Networks — SASN 2007*, volume 4572 of *Lecture Notes in Computer Science*, pages 73–86. Springer Verlag, 2007.
28. C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 38(21):1831–1832, Oct. 1999.
29. C. D. Walter. Simple power analysis of unified code for ECC double and add. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 191–204. Springer Verlag, 2004.
30. C. D. Walter and S. Thompson. Distinguishing exponent digits by observing modular subtractions. In D. Naccache, editor, *Topics in Cryptology — CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 192–207. Springer Verlag, 2001.
31. T. Yanık, E. Savaş, and Ç. K. Koç. Incomplete reduction in modular arithmetic. *IEE Proceedings – Computers and Digital Techniques*, 149(2):46–52, Mar. 2002.
32. Y. Zhang and J. Großschädl. Efficient prime-field arithmetic for elliptic curve cryptography on wireless sensor nodes. In *Proceedings of the 1st International Conference on Computer Science and Network Technology (ICCSNT 2011)*, volume 1, pages 459–466. IEEE, 2011.