Efficient and Side-Channel Resistant RSA Implementation for 8-bit AVR Microcontrollers

Zhe Liu*†, Johann Großschädl†, and Ilya Kizhvatov†

*School of Computer Science and Technology, Shandong University, Jinan, China
sduliuzhe@gmail.com, zhe.liu.001@student.uni.lu
†Laboratory of Algorithmics, Cryptology and Security, University of Luxembourg
{johann.groszschaedl,ilya.kizhvatov}@uni.lu

***Abstract*—The RSA algorithm is the most widely used public-key cryptosystem today, but difficult to implement on embedded devices due to the computation-intense nature of its underlying arithmetic operations. Different techniques for efficient software implementation of the RSA algorithm have been proposed; these range from high-level approaches, such as exploiting the Chinese Remainder Theorem (CRT), down to smart optimizations of the low-level modular arithmetic (e.g. hybrid multiplication). In the present paper we introduce a new variant of the hybrid method for multiple-precision multiplication that optimizes both memory accesses and register allocation. The inner loop of our improved hybrid method saves about 7.8% in execution time compared to the original one of Gura et al. We combine our hybrid method with the Separated Operand Scanning (SOS) Montgomery multiplication into the HSOS method, a new technique to perform long-integer modular arithmetic. Our practical results, obtained on an ATmega128 microcontroller, show that the HSOS method outperforms other modular multiplication techniques for typical operand lengths used in RSA. A 1024-bit private-key operation can be carried out in less than $76 \cdot 10^6$ clock cycles when taking advantage of the CRT and *m*-ary exponentiation method, which represents a new speed record for RSA on 8-bit controllers. We also protected our RSA implementation against power analysis attacks via the integration of low-cost countermeasures. These countermeasures increased the execution time of the private-key operation by just 12% compared to an unprotected version.**

***Index Terms*—Lightweight implementation, multiple-precision arithmetic, AVR architecture, side-channel cryptanalysis, countermeasures against side-channel attacks.**

I. Introduction

The Rivest-Shamir-Adleman (RSA) algorithm, developed in the 1970s [21], is the foundation of the most important public-key cryptosystem in use today. Both public-key encryption and digital signature schemes can be constructed on basis of the RSA algorithm. The security of the RSA cryptosystem relies on the (presumed) intractability of the Integer Factorization Problem (IFP). At present, in order to achieve a proper level of security, the modulus N of an RSA cryptosystem should have a size of at least 1024 bits. The costly part of RSA (and other public-key schemes) is modular exponentiation, i.e. an operation of the form $C = M^E \bmod N$, whereby in our case the operands have a length of 1024 bits. An exponentiation of integers of such size can be easily done on commodity PCs and laptops, but may result in unacceptably long delays on smart cards, sensor nodes, and other embedded devices with modest processing power. Elliptic Curve Cryptography (ECC) [3] is generally seen as a viable alternative to RSA due to its relatively small key sizes (e.g. 160 bits vs. 1024 bits) and the resulting savings in execution time and memory footprint, all of which are relevant in the embedded domain.

Despite the ongoing proliferation of ECC, RSA is still the most important (and most widely used) public-key algorithm in the world. In particular, RSA is an integral part of modern security protocols such as SSL/TLS, WTLS, and IKE. On the other hand, ECC has found adoption in security architectures for ad-hoc and sensor networks [16]. However, the Internet is dominated by RSA-based certificates and the corresponding Public-Key Infrastructure (PKI). VeriSign, a big international Certification Authority (CA), prefers RSA-based certificates over their ECC counterparts for SSL, even when the clients are resource-constrained devices like PDAs or mobile phones [24]. The main reason is the fact that secure SSL connections for Web applications (e.g. e-banking) are usually established such that the server is authenticated to the client, but not vice versa. Client authentication is typically done at the application layer (and not the SSL layer), e.g. by a user entering his ID and a password. Negotiating an RSA-based SSL connection with server-only authentication is relatively inexpensive on the client side since all RSA operations are executed with public exponents, which are generally small. However, this picture ceases to be valid for the "Internet of Things," in which the clients are objects such as sensor nodes or RFID tags instead of conventional computing platforms controlled by a user. In this setting, client authentication can not be done by entering a password, but must be performed via private-key operations involving full-size (i.e. 1024-bit) exponents. Therefore, efficient RSA implementations for embedded devices play a vital role in the expansion of well-established security protocols to the Internet of Things.

There exists a considerable literature on efficient software implementation of the RSA algorithm [11]. Quisquater and Couvreur [20] were the first to demonstrate that the Chinese Remainder Theorem (CRT) can be utilized to accelerate the private-key operations of RSA by a factor of nearly four. The efficiency of different exponentiation algorithms (e.g. square and multiply method, *m*-ary technique, window method) was studied in [10], [11]. Montgomery [19] proposed an ingenious

technique for modular reduction that avoids the trial division and performs simple shift operations instead. Several variants of Montgomery multiplication [19], including the "Finely Integrated Product Scanning" (FIPS) method, were analyzed in [12]. The Karatsuba-Comba-Montgomery (KCM) method [6] is another variant that combines Karatsuba and Comba-style multiplication techniques with Montgomery reduction. Gura et al [7] introduced the hybrid strategy for multiple-precision multiplication, which reduces the number of memory accesses (i.e. loads) on processors with a large register file.

In the present paper, we describe our efforts to develop a high-speed RSA software implementation for the AVR family of microcontrollers [1], in particular the ATmega128 [2]. The AVR is one of the most widely used 8-bit RISC architectures and microcontrollers implementing this instruction set can be found in various smart cards and sensor nodes. Our work is motivated by the fact that there exist very few publications on RSA implementations for 8-bit controllers, whereas numerous papers have been written about efficient ECC software for the ATmega128 (see e.g. [15], [16] and the references therein). In addition, we were interested to determine which performance well-written RSA software can achieve on an "8-bitter." The RSA implementation we introduce in this paper requires less than $76 \cdot 10^6$ clock cycles to execute a full 1024-bit private-key operation on an ATmega128, which sets a new speed record for RSA on 8-bit microcontrollers. Certain AVR models have a nominal clock frequency of 32 MHz (e.g. XMEGA), hence this cycle count corresponds to an execution time of merely 2.5 seconds. Our high-speed RSA implementation advances the state-of-the-art in two main aspects. First, we developed a new variant of Gura et al's hybrid method with an improved inner loop operation that saves up to 7.8% in execution time compared to the original one from [7]. Our loop is similar to that of Lederer et al [15], but loads 4 bytes of each operand per iteration (instead of just 2), which is possible thanks to a revised register allocation strategy. Second, we employed the Separated Operand Scanning (SOS) method [12] for modulo multiplication since we found that it can be better combined with our hybrid technique than both the FIPS method and the KCM method.

We also protected our RSA software against side-channel attacks [17], in particular Simple Power Analysis (SPA) and Differential Power Analysis (DPA). Both belong to the genre of implementation attacks and use information leaking from a device while it executes a cryptographic algorithm to reveal the secret key. SPA refers to a scenario where an attacker has to collect only one, or very few, power traces and attempts to recover the key by focusing on differences between patterns within a trace. In contrast, DPA uses several or many traces and analyzes differences between the traces. There exists an abundant literature on countermeasures against these attacks [17]. For example, an RSA implementation can be protected against SPA attacks by eliminating "irregularities" in the exponentiation algorithm. In the case of the m-ary method, this is typically achieved by converting the m-ary expansion of the exponent into an alternative representation that is based on a digit set not containing zero. Numerous exponent conversion techniques facilitating a regular implementation of the m-ary method are described in [8]. However, we decided to not use such a conversion and implemented a simple countermeasure that, to our knowledge, has not appeared before in the open literature. Our idea is to represent the multiplicative identity of $\mathbb{Z}_N$ (where N is the modulus) by the Montgomery image of 1, which is simply the residue $R \bmod N = 2^n \bmod N$. The quantity R is called Montgomery radix and usually selected as a power of 2 that is larger than N, e.g. $R = 2^n$ for a modulus of bitlength n [6], [11]. In this case, the Montgomery image of 1 (i.e. $2^n \bmod N$) can be calculated by subtracting N from 2^n. A multiplication by the Montgomery image of 1 carried out during the execution of the m-ary exponentiation method does not leak any SPA-relevant information, as will be shown in Section III-C.

II. Preliminaries

RSA operations are modular exponentiations of very large integers with a typical size of between 512 to 2048 bits. The large integers are, in general, represented by arrays of single-precision words (i.e. arrays of bytes in our case since we are implementing RSA for an 8-bit microcontroller). We denote long integers by uppercase letters and use the corresponding lowercase letters for the individual bytes. A concrete example is $A = (a_{s-1}, \ldots, a_1, a_0)$ with $0 \leq a_i < 2^8$, whereby s refers to the number of bytes. The following subsections summarize a number of basic algorithms for long-integer arithmetic.

A. Basic Multiplication Techniques

The simplest algorithm for multiplying large integers is the so-called *schoolbook method*, which is often also referred to as *operand-scanning method* [6], [10], [11]. In essence, this algorithm has a "nested-loop" structure with an inner loop in which operations of the form $(u,v) \leftarrow a \cdot b + c + d$ are carried out, i.e. two bytes are multiplied and another two bytes are added to the 16-bit product. The final result of this operation is at most 16 bits long when a, b, c, d are bytes. Hence, the result can be stored in (u,v), a pair of 8-bit registers [6]. The schoolbook algorithm iterates the inner loop exactly s^2 times when each of the two operands consists of s bytes. This means that, in total, s^2 `mul` instructions have to be executed for the multiplication of two s-byte operands. Each iteration of the inner loop also performs two `ld` (load), four `add` (resp. `adc`) and one `st` (store) instruction (see [6] for a detailed analysis of the total instruction counts). A characteristic feature of the schoolbook is that, in the inner loop, a byte of operand B is multiplied by all the bytes of operand A, i.e. the 2-byte partial products are processed in a row-wise fashion.

A different (but typically more efficient) algorithm for the multiplication of long integers was introduced by Comba in [4]. *Comba's method* (also called *product-scanning method*) processes the 2-byte partial products in a column-wise fashion instead of row-by-row as the schoolbook method. It features a nested loop structure (as the schoolbook method), but has two outer loops and two inner loops. Nonetheless, the total

number of iterations of the inner loop is the same as for the schoolbook method, namely s^2 for two s-byte operands. The operation executed in the two inner loops is a multiply-accumulate operation of the form $(t,u,v) \leftarrow (t,u,v) + a \cdot b$, i.e. two bytes of the operands are multiplied and the 16-bit product is added to a running sum held in (t,u,v), a set of three registers [6]. Note that, when performing a number of iterations of the inner loop, several partial products are added to a so-called column sum, which means that this sum will get longer than two bytes, i.e. three registers are needed to store it. Comba's method executes one `mul`, two `ld`, and three `add` (resp. `adc`) instructions per iteration of the loop [6]. However, Comba's method does not require `st` instructions in the inner loop as the bytes of the final product are written back to memory in the outer loop.

Karatsuba [9] introduced a multiplication algorithm that is asymptotically faster than both the schoolbook and Comba's method. The basic idea of Karatsuba's multiplication method is to split a multiplication of two s-byte operands into to three multiplications of size $s/2$, which is possible at the expense of some overhead, e.g. an increased number of additions. The three half-size multiplications can be accomplished using the schoolbook method (operand scanning) or Comba's method (product scanning). In summary, Karatsuba's method executes only $3s^2/4$ `mul` instructions to multiply two s-byte operands [6]. However, as mentioned before, this approach incurs some overhead; therefore, it is only attractive for the multiplication of relatively large operands.

B. Hybrid Multiplication

Hybrid multiplication was first introduced by Gura et al in [7]. The hybrid technique is, strictly speaking, not a new multiplication algorithm, but rather a sophisticated optimization of the product-scanning approach. It combines the advantages of the schoolbook method (operand scanning) and Comba's method (product scanning), and aims at reducing the overall number of memory accesses (especially `ld` instructions) at the expense of increased register usage. Consequently, the hybrid multiplication technique is only beneficial on general-purpose processors featuring a large number of registers, which is the case for the ATmega128.

A standard implementation of the product-scanning method results in an inner loop in which one byte of operand A and one byte of operand B are loaded from memory, multiplied together, and added to a column sum held in three registers [6]. The main idea of the hybrid multiplication method is to process $d > 1$ bytes of the two operands A and B in a single iteration of the loop, which reduces the number of iterations by a factor of d. Furthermore, the hybrid method reduces the number of `ld` instructions since the d bytes of the operands are used several times (i.e. for several `mul` instructions) in the inner loop, but loaded only once. Gura et al employ Comba's method as the "outer algorithm" and the schoolbook method as "inner algorithm." More precisely, they compute a Comba multiplication that consists of partial products obtained by the schoolbook method (see [7] for further details).

C. Montgomery Modular Multiplication

Montgomery introduced in [19] a very efficient algorithm for a modular reduction of the form $Z = P \cdot 2^{-n} \bmod N$. This so-called Montgomery reduction is usually combined with a multiplication of two operands, i.e. $P = A \cdot B$, into an operation of the form $Z = A \cdot B \cdot 2^{-n} \bmod N$, commonly referred to as Montgomery multiplication [11], [12]. It is possible to perform a Montgomery multiplication in an interleaved or a separated fashion using either operand scanning or product scanning. In this subsection, we overview three well-known approaches to execute a Montgomery multiplication: the Separated Operand Scanning (SOS) method, the Finely Integrated Product Scanning (FIPS) method, and the Karatsuba-Comba-Montgomery (KCM) method. The former two methods are described and analyzed in [12], whereas further details on the KCM method can be found in [6]

As its name suggests, the SOS method completely separates multiplication and modular reduction steps, i.e. the reduction is done after the product $A \cdot B$ has been entirely computed. In the original description of the SOS method in [12], Koç et al performed both the multiplication and Montgomery reduction according to the schoolbook method. However, one can also use the product-scanning technique for both operations. When realized on basis of the schoolbook method, the inner loops of both the multiplication and Montgomery reduction execute an operation of the form $(u,v) \leftarrow a \cdot b + c + d$ as explained in Subsection II-A and [6]. Assuming s-byte operands, the SOS technique executes s^2 `mul` instructions for the multiplication and $s^2 + s$ `mul` instructions for the reduction, which amounts to $2s^2 + s$ `mul` instructions altogether.

The FIPS technique performs multiplication and reduction steps in an interleaved fashion in the same inner loop. From an algorithmic point of view, the FIPS method has a nested loop structure with two inner loops, very similar to Comba's method. In each iteration of the inner loop, two multiply-accumulate operations of the form $(t,u,v) \leftarrow (t,u,v) + a \cdot b$ are carried out; one contributes to the computation of $A \cdot B$, and the second to the Montgomery reduction. The operation in the inner loop of the FIPS technique is identical to one of the Comba method as described in Subsection II-B. In total, the CIOS method executes exactly $2s^2 + s$ `mul` instructions when the operands consist of s bytes. Even though FIPS and SOS execute exactly the same number of `mul` instructions, the total number of `add` (resp. `adc`) and `st` instructions differs.

Contrary to FIPS, the KCM method completely separates the multiplication of A by B and the reduction of the obtained product modulo N [6]. The KCM method uses a combination of Karatsuba and Comba multiplication for the former, while the latter is realized with a product-scanning technique as in Algorithm 1. Since the multiplication is based on Karatsuba's algorithm, the KCM method is asymptotically faster than the SOS and FIPS method. The total number of `mul` instructions executed by the KCM technique is only $\frac{7}{4}s^2 + s$ for operands consisting of s bytes. However, the saving in `mul` instructions comes at the cost of an increased number of `add` (resp. `adc`)

Algorithm 1. Montgomery reduction (product scanning form)

Input: An s-byte modulus $N = (n_{s-1}, \ldots, n_1, n_0)$, a product P in the range of $[0, 2N-2]$, pre-computed constant $n'_0 = -n_0^{-1} \bmod 2^8$.
Output: Montgomery residue $Z = P \cdot 2^{-n} \bmod N$.

$(t,u,v) \leftarrow 0$
for i from 0 by 1 to $s-1$ **do**
  for j from 0 by 1 to $i-1$ **do**
    $(t,u,v) \leftarrow (t,u,v) + z_j \cdot n_{i-j}$
  end for
  $(t,u,v) \leftarrow (t,u,v) + p_i$
  $z_i \leftarrow v \cdot n'_0 \bmod 2^8$
  $(t,u,v) \leftarrow (t,u,v) + z_i \cdot n_0$
  $v \leftarrow u$, $u \leftarrow t$, $t \leftarrow 0$
end for
for i from s by 1 to $2s-2$ **do**
  for j from $i-s+1$ by 1 to $s-1$ **do**
    $(t,u,v) \leftarrow (t,u,v) + z_j \cdot n_{i-j}$
  end for
  $(t,u,v) \leftarrow (t,u,v) + p_i$
  $z_{i-s} \leftarrow v$
  $v \leftarrow u$, $u \leftarrow t$, $t \leftarrow 0$
end for
$(t,u,v) \leftarrow (t,u,v) + p_{2s-1}$
$z_{s-1} \leftarrow v$, $z_s \leftarrow u$
if $Z \geq N$ **then** $Z \leftarrow Z - N$ **end if**

instructions. On the other hand, the KCM technique performs only $10s$ `st` instructions as both the Karatsuba-Comba method and Algorithm 1 implement a product-scanning approach (see [6] for further details).

D. Modular Exponentiation

The m-ary method, as described in [10] and [11], is based on m-ary expansion of the exponent E. We mainly focus on our SPA-resistant variant of the m-ary method, which will be evaluated and analyzed in Section III-C. Our variant divides the n-bit exponent E into $s = \lceil n/4 \rceil$ digits e_i, each consisting of four bits (as shown in Algorithm 2). The rationale behind this choice of $m = 2^4 = 16$ is to balance memory requirements and performance (i.e. number of modular multiplications).

Algorithm 2 works in the following way: we first generate a look-up table containing $2^4 = 16$ entries through squarings and multiplications (we can see this look-up table as a database for speeding up the modular exponentiation). Then, we start processing at e_{s-1}, the most-significant 4-bit digit of the exponent E, and assign the corresponding table entry as the initial value of C, which also serves as input for our main loop. The main loop starts with e_{s-2}, the second digit of the exponent E, and works downwards. For each 4-bit digit, we do four Montgomery squarings, then look up the table to find the value of $\mathrm{T}[e_i]$ (where e_i is the i-th digit of E) and finally do the Montgomery multiplication.

The difference between Algorithm 2 and a straightforward implementation of the m-ary technique (such as presented in [10] or [11]) is that we do not treat zero digits in a special way. As outlined in Section I, we represent the multiplicative identity of $\mathbb{Z}_N$ through the Montgomery image of 1 (which is $2^n - N$ in our case) and perform the main loop without any conditional statements. Zero digits of E are processed in the same way as other (i.e. non-zero) digits and, therefore, no SPA-relevant information is leaking from the loop, provided that the Montgomery multiplication and squaring operations are also implemented in an SPA-resistant way. However, the drawback of our SPA countermeasure is that it works only in combination with Montgomery arithmetic.

Algorithm 2. SPA-resistant m-ary method ($m = 16$)

Input: An n-bit modulus N, a message M in the range of $[0, N-1]$, an exponent E represented by $s = \lceil n/4 \rceil$ digits e_i with $0 \leq e_i < 16$ for $i = 0 \ldots s-1$, i.e. $E = (e_{s-1}, \ldots, s_1, s_0)$.
Output: $C = M^E \bmod N$.

{ Generate the look-up table T[0..15]}
T[0] = $2^n - N$ { Montgomery image of 1}
T[1] = M
Calculate T[2] = M^2, T[4] = M^4, and T[8] = M^8 via squaring
Calculate the remaining table entries via multiplication
$C = \mathrm{T}[e_{s-1}]$
for i from $s-2$ by 1 down to 0 **do**
  $C = C^{16} \bmod N$ { Four modular squarings}
  $C = C \cdot \mathrm{T}[e_i] \bmod N$ { One modular multiplication}
end for

Our variant of the m-ary method with $m = 2^4 = 16$ always performs exactly $n/4$ multiplications and n squarings for an n-bit exponent E, independent of its Hamming weight. The memory footprint of our m-ary method is bearable (e.g. when taking 1024-bit RSA as example, the look-up table occupies 2048 bytes in RAM, or 1024 bytes when using the CRT).

III. Our RSA Implementation

As mentioned in Section I, our RSA implementation profits from a new variant of the hybrid method and the combination of this new hybrid method with the SOS technique for Montgomery multiplication. The resulting Hybrid-SOS (or HSOS for short) technique was prototyped on an ATmega128 [2], a low-power 8-bit microcontroller based on the AVR instruction set architecture [1]. The ATmega128 has 128 kB of in-system re-programmable flash, 4 kB internal SRAM, and features a total of 32 general-purpose registers.

A. Improved Hybrid Method

The execution time of the Comba method (as well as the performance of all modular multiplication techniques based on it, e.g. FIPS and KCM) can be significantly improved when balancing between register usage and the number of memory accesses. Gura's hybrid multiplication technique [7] aims to combine the individual advantages of operand scanning and product scanning; it uses Comba's method as the *outer* algorithm and the schoolbook method as the *inner* algorithm. In other words, the hybrid method obtains the product according to Comba's technique, whereby the d^2 partial products in the inner loop are generated and summed up in the same way as in the schoolbook method. The saving in memory accesses is due to the fact that $d > 1$ bytes of the operands are processed in each iteration of the inner loop, but loaded from memory only once. In order to make full use of the large register file of the ATmega128, one typically chooses $d = 4$, which means

that four bytes of operand A and four bytes of operand B are processed in each iteration of the inner loop as described in [7]. Consequently, four bytes of each A and B are loaded from memory into eight registers, multiplied in a row-wise fashion (following the schoolbook method), and added to a column sum held in nine general-purpose registers as depicted on the left of Figure 1. This operation requires eight load (`ld`) and 16 `mul` instructions; furthermore, some 48 `add`/`adc` as well as 16 `mov` instructions (or, alternatively, 64 `add`/`adc` instructions) are executed when $d = 4$ (see [7] for further details).

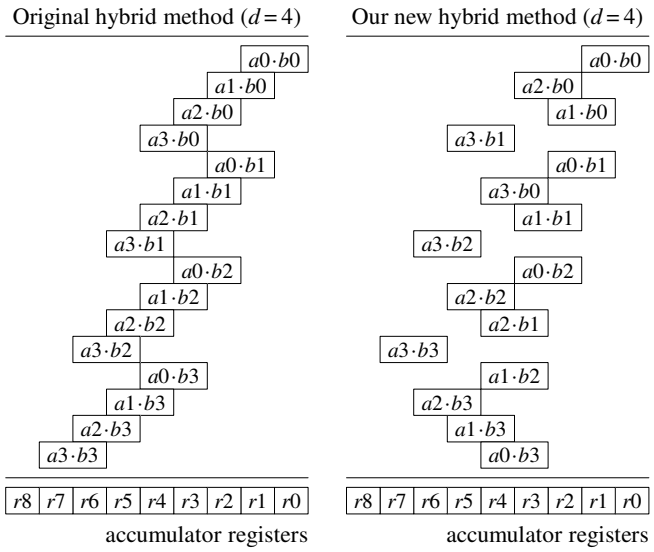

Fig. 1. Comparison of the inner loop operation of the original hybrid method (left) and our improved variant (right) for $d = 4$.

Our new variant of the hybrid method aims at reducing the overall number of `mov` (or `movw`) instructions executed in the inner loop compared to Gura's original method. To achieve this, we schedule the `mul` instructions in a non-conventional order in our inner loop such that their addition to the column sum (including carry propagation) can be performed in "one pass" for several partial products. The right part of Figure 1 depicts an example for $d = 4$ that illustrates the difference to Gura's method. This approach for implementing the hybrid method is originally due to Lederer et al [15], but we found a better register allocation strategy that allows us to process four bytes of each operand at a time, whereas Lederer et al could only process two bytes. Figure 1 shows as example the multiplication of the 4-byte word $A = (a_3, a_2, a_1, a_0)$ by the 4-byte word $B = (b_3, b_2, b_1, b_0)$. We first multiply a_0 by b_0 and a_2 by b_0, and then move the obtained 2-byte partial products to four temporary registers with help of the `movw` instruction [1]. After multiplying a_1 by b_0, we add the partial product to the content of the four temporary registers, which takes one `add` and two `adc` instructions. Note that the second `adc` can not produce a "carry-out," as is explained in [23]. Next, we multiply a_3 by b_1 and then add the four temporary registers along with the partial product $a_3 \cdot b_1$ to the content of the nine accumulator registers, which can be done in "one pass." In total, the addition of $a_0 \cdot b_0$, $a_2 \cdot b_0$, $a_1 \cdot b_0$, and $a_3 \cdot b_1$ to the column sum requires only 12 `add` (resp. `adc`) as well as two `movw` instructions. The next group of four partial products in Figure 1 (i.e. $a_0 \cdot b_1$, $a_3 \cdot b_0$, $a_1 \cdot b_1$, and $a_3 \cdot b_2$) is processed in the same way; this also takes two `movw` instructions but only 11 `add` (resp. `adc`). Thereafter, the third group of four partial products, namely $a_0 \cdot b_2$, $a_2 \cdot b_2$, $a_2 \cdot b_1$, and $a_3 \cdot b_3$, is added to the column sum using 10 `add`/`adc` instructions in total. The last group of four partial products is summed up in a slightly different way. First, the partial products $a_1 \cdot b_2$ and $a_2 \cdot b_3$ are moved to the four temporary registers. The next two partial products (i.e. $a_1 \cdot b_3$, $a_0 \cdot b_3$) are added to the content of the four temporary registers, which requires seven `add` (or `adc`) instructions, but can not produce a "carry-out" [23]. Now the temporary registers are added to the column sum held in nine registers. In summary, 46 `add`/`adc` instructions are executed for a 4-byte by 4-byte multiplication and the addition of the product to nine registers. Moreover, our hybrid method has to execute eight `movw` instructions.

Our variant of the hybrid technique requires four registers for temporary storage of two 2-byte partial products. On the other hand, the specific order in which we multiply the bytes of the operands means that we do not necessarily need to have all four bytes of B in registers at the same time, i.e. we can load them in 2-byte portions, using only two registers. More precisely, at the beginning of each loop iteration, we load all four bytes of operand A, but only the bytes b_0, b_1 of B. Then we perform the first six byte multiplications as shown on the right side of Figure 1; these multiplications only need b_0 and b_1, but not b_2 and b_3. After the the sixth multiplication (in which $a_3 \cdot b_0$ is produced), the byte b_2 is loaded and placed in the register holding b_0 (we can simply overwrite b_0 as it is not needed anymore). We proceed with the next five multiplications; after computing the product $a_2 \cdot b_1$, we load b_3 and overwrite b_1 (this is possible since b_1 is not needed anymore at this time). In summary, our variant of the hybrid method requires six registers for the bytes of the operands, as well as four registers for temporary storage of partial products. Note that temporary storage of partial products (and also execution of `movw` instructions) is not necessary on processors equipped with a multiplier that does not modify the carry flag.

TABLE I
COMPARISON OF INSTRUCTION COUNTS ON THE ATMEGA128

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 [7]	1360	400	167	40	355	197	3106
Uhsadel [23]	986	400	238	40	355	184	2881
Scott [22]	1263	400	200	40	70	38	2651
Our work	1194	400	200	40	212	179	2865

Table I summarizes instruction counts and total execution time (in clock cycles) of our improved hybrid method for a (160×160)-bit multiplication on an ATmega128 processor. We use (160×160)-bit multiplication as a benchmark in order to allow for direct comparison with previous work that targeted

elliptic curve cryptography [3] instead of RSA. Note that the instruction counts in the columns labeled with `add`, `ld`, and `mov` also include `adc`, `ldd`, and `movw`, respectively (i.e. we do not distinguish between `add` and `adc` as they both require one cycle on AVR processors). Our variant of the hybrid method executes a (160×160)-bit multiplication in 2865 clock cycles on the ATmega128, which is approximately 7.8% faster than the original hybrid method of Gura et al [7]. This saving in execution time is mainly due to the fact that we perform only 212 `mov` (resp. `movw`) instructions, whereas Gura et al require some 355 `mov` or `movw` instructions. Furthermore, our special scheduling of the multiplications in the inner loop reduces the number of `add` (resp. `adc`) instructions, as was noticed before in [15]. The hybrid variant of Uhsadel et al [23] takes 2881 cycles, even though their implementation (as well as the one of Gura [7]) is based on $d = 5$ for 160-bit operands (instead of $d = 4$ as in our work), i.e. five bytes of each operand are processed per loop iteration. However, our hybrid method is a little slower than Scott et al's implementation [22], mainly because they fully unrolled the loops, which allowed them to yield further savings at the expense of larger code size. Full loop unrolling may be a viable optimization in elliptic curve cryptography, but not for RSA.

B. Hybrid Montgomery Multiplication

We integrated our new hybrid technique as inner loop into different Montgomery multiplication and squaring algorithms such as SOS, FIPS, and KCM (all of which are described in Subsection II-C). Furthermore, we developed a small Assembly library containing basic long-integer arithmetic operations (addition, subtraction, comparison, and so on). The addition and subtraction of two operands A, B are implemented via a simple loop that iterates through the individual bytes of these operands. However, in all these basic operations, we process four bytes of the operands "at once" (i.e. per iteration of the loop), starting with the least significant 4-byte block. Also the different Montgomery multiplication and squaring algorithms are implemented in Assembly language and highly optimized for speed. On the other hand, the modular exponentiation is written in ANSI C and calls the low-level Assembly functions for modular multiplication and squaring. In the following, we describe in detail our implementation of three hybrid variants for Montgomery multiplication: hybrid SOS (HSOS), hybrid FIPS (HFIPS), and hybrid KCM (HKCM).

Hybrid SOS (HSOS): As mentioned in Subsection II-C, the SOS method separates multiplication and modular reduction steps, i.e. the reduction is done *after* the full product $A \cdot B$ has been computed. The original description of the SOS method in [12] uses the schoolbook approach (i.e. operand scanning) for both the multiplication and Montgomery reduction. However, in order to take advantage of the hybrid technique, we perform the multiplication according to Comba's method in combination with our improved inner loop as described in the previous subsection, whereas the reduction is implemented on basis of Algorithm 1, but using our improved inner loop. The Comba multiplication (as well as the Montgomery reduction) process four bytes of each operand per loop iteration.

Hybrid FIPS (HFIPS): As indicated by its name, the HFIPS method combines our improved hybrid technique with FIPS Montgomery multiplication. A specific property of FIPS and HFIPS is that they perform multiplication and reduction steps in an interleaved fashion in the same inner loop. HFIPS uses our improved hybrid technique as "inner algorithm" and the product scanning method as "outer algorithm." Each iteration of the inner loop performs two multiplications of 4-byte by 4-byte words and adds the two resulting 8-byte products to the column sum, which is held in nine registers. Of course, this processing of four bytes of the operands per iteration (instead of just a single byte as in standard FIPS) reduces the overall number of loop iterations by a factor of four.

Hybrid KCM (HKCM): HKHM is our variant of the KCM multiplication; it is based on the improved hybrid method as inner loop. Like KCM, also HKHM completely separates the multiplication and reduction operation; the former is realized using Karatsuba-Comba multiplication and the latter via the product-scanning technique of Algorithm 1. We implemented the HKCM multiplication as a C function with calls to the low-level Assembly routines for performing multiple-precision arithmetic. Firstly, we divide the multiplication of two s-byte operands into three $(\frac{s}{2})$-byte multiplications using Karatsuba's method, and then perform these "half-length" multiplications using Comba's method combined with our optimized hybrid technique. Then, the Montgomery reduction is carried out as specified in Algorithm 1, but again with using our improved hybrid method in the inner loop(s).

Optimizations for Squaring: The square A^2 of a multiple-precision integer A can be computed significantly faster than the product $A \cdot B$ of two distinct integers. Due to a symmetry in the squaring operation, many partial products appear twice since $a_i \cdot a_j = a_j \cdot a_i$. All partial products $a_i \cdot a_j$ for $i \neq j$ need to be computed only once and can be simply left-shifted to be doubled. Compared to standard Comba multiplication, the optimized Comba squaring saves some 20% execution time for 256-bit operands. Optimizations for Montgomery squaring can be easily integrated into both HSOS and HKCM, but are rather complicated for HFIPS.

TABLE II
EXECUTION TIME OF HSOS, HFIPS, AND HKCM (IN CLOCK CYCLES).

Algorithm	256 bit	512 bit	768 bit	1024 bit
HSOS ($d = 4$)	18655	65649	141585	246462
HFIPS ($d = 4$)	19727	70592	153007	266975
HKCM ($d = 4$)	20994	66142	135901	232450

Analysis and Comparison: In order to analyze and compare these three hybrid Montgomery multiplication techniques, we simulated their execution time using operands ranging from 256 to 1024 bits; the results (in clock cycles, not including a potentially necessary final subtraction) are summarized in Table II. The HSOS method outperforms HFIPS at all operand

lengths, which seems quite surprising at a first glance given that HFIPS merges multiplication and reduction into a single inner loop and, thus, executes the loop overhead (i.e. update of a loop counter, branch instruction) only once. However, a disadvantage of HFIPS is the need to maintain three pointers to operands (A, B, N) as well as a pointer to the result in the inner loop, which is not possible in our implementation since we do not have sufficient free registers to hold four pointers (each of which occupies two registers). Therefore, we need to `push` and `pop` one of these four pointers to/from the stack in each iteration of the loop, which costs four clock cycles. On the other hand, the overhead in the execution of an iteration of the loop (i.e. increment or decrement of a loop counter and branch instruction) amounts to three cycles. Consequently, a separation of multiplication and reduction steps (as in HSOS) is more efficient than the "fusion" of these operations into a single inner loop (like in HFIPS), mainly due to the massive register usage of the hybrid method.

The HKCM method is faster than HSOS (and also HFIPS) for operands exceeding 512 bits in length, but slower in the case of 256-bit operands. Since HKCM is based on Karatsuba multiplication, it is asymptotically faster than both HSOS and HFIPS, but involves also a certain overhead (e.g. additions to combine the half-size products), which explains its moderate performance for 256-bit operands. For 512-bit operands, the HKCM method achieves essentially the same performance as HSOS, and both are roughly 7% faster than HFIPS.

C. Performance Evaluation of 1024-bit RSA

We also implemented an entire RSA private-key operation (i.e. decryption, signature generation) using the CRT and the m-ary exponentiation method. As mentioned in Section I, the CRT allows one to speed up RSA private-key operations by a factor of almost four. The basic idea is to split up a full-size exponentiation of the form $M = C^D \bmod N$ into two half-size exponentiations, whereby one is performed modulo P and the other modulo Q [20]. Of course, this is only possible when the prime factors P and Q of N are known. The CRT requires some pre-processing (to split the ciphertext C and decryption exponent D into two half-size parts) and post-processing (to combine the results of the two half-size exponentiations into M). Both the pre- and postprocessing involve a few operands that depend only on P and Q, i.e. they are constant as long as the key does not change. We pre-computed these constants to allow for an efficient execution of CRT-based decryption (resp. signature generation).

Our implementation performs the half-size exponentiations using the m-ary exponentiation method with $m = 2^4$ as shown in Algorithm 2. This version of the m-ary method processes a 4-bit digit of the exponent at a time, which means it performs n squarings and $n/4$ multiplications to obtain the result of an n-bit modular exponentiation. In contrast, the simple square-and-multiply technique requires about $n/2$ multiplications on average, but n multiplications in the worst case. Therefore, the m-ary method with $m = 2^4$ can save some 15% in execution time on average when taking into account that a squaring is slightly faster than a multiplication. However, in a worst case situation (i.e. the exponent has a Hamming weight of 1), the m-ary method is significantly faster. The m-ary exponentiation with $m = 2^4$ requires pre-computation and storage of a table containing 16 powers of the base, which occupies 1024 bytes in RAM for 512-bit operands.

TABLE III
PERFORMANCE OF 1024-BIT RSA IMPLEMENTATIONS

Implementation	Cycle count	Notes
Deng et al [5]	$60.0 \cdot 10^6$	C, public-key operation
Watro et al [28]	$58.0 \cdot 10^6$	C, public-key operation
Gura et al [7]	$87.92 \cdot 10^6$	C and assembly, CRT
Wander et al [26]	$88.0 \cdot 10^6$	C and assembly, CRT
Wang et al [27]	$172.0 \cdot 10^6$	C and assembly, CRT
Our work	$75.68 \cdot 10^6$	C and assembly, CRT

The execution time of a 1024-bit RSA private-key operation is roughly $75.68 \cdot 10^6$ clock cycles on the ATmega128 when exploiting the CRT and using the m-ary method for the two 512-bit exponentiations. We implemented the m-ary technique on basis of the HSOS method for Montgomery multiplication and squaring since it is slightly faster than the HKCM method and can also be easier protected against SPA attacks. A cycle count of $75.68 \cdot 10^6$ corresponds to an execution time of less than five seconds when the ATmega128 is clocked with the maximum frequency of 16 MHz. Table III compares our execution time with that of previously-published implementations of 1024-bit RSA operations on the ATmega128. Both Watro et al [28] and Deng et al [5] specify only the execution time of a 1024-bit RSA public-key operation in their papers. The results presented in Table III show that our work outperforms previous implementations on the ATmega128 and establishes a new speed record for 1024-bit RSA on 8-bit processors. We attribute the performance gain to our improved variant of the hybrid method and the new HSOS technique for Montgomery multiplication and squaring.

IV. Low-Cost Side-Channel Countermeasures

In the following, we first demonstrate that countermeasures against Simple Power Analysis (SPA) [14] are necessary in an RSA implementation. Then, we show that an implementation with SPA countermeasures may still succumb to Differential Power Analysis (DPA), and describe further countermeasures to protect RSA operations against DPA.

A. SPA Attack and Countermeasure

To show vulnerability of the standard square-and-multiply algorithm, we attack our implementation of this algorithm via SPA. The square-and-multiply method is fairly simple; it uses the binary representation of the exponent E and computes the result as a sequence of modular multiplications and squarings [10]. Our implementation of the square-and-multiply method starts at most significant non-zero bit of E and works downwards. Whenever the current bit e_i of E is 1, a squaring and a multiplication are performed. However, if $e_i = 0$, only the squaring operation is carried out. Our practical experiments

show that multiplications and squarings take different time and exhibit different patterns in a side-channel trace as they consume different amounts of power during execution. Figure 2 shows as example the EM and power profile of a squaring and a multiplication. The right part of the EM (resp. power) trace, corresponding to multiplication, is clearly distinct from the left part, corresponding to squaring. Both the power and EM trace exhibit more significant peaks in the multiplication than in the squaring. These peaks make it relatively easy to obtain the bits of the secret RSA exponent directly from the power or EM trace.

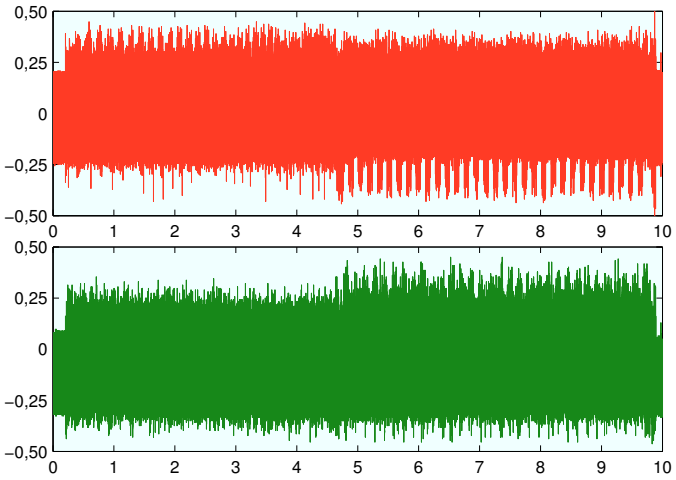

Fig. 2. EM (top) and power (bottom) traces of an ATmega128 executing a squaring followed by a multiplication; no averaging performed.

Moreover, as described in [25], the Montgomery reduction is vulnerable to SPA because of the conditional subtraction of the modulus N that is executed to keep the result bounded within the range of $[0, N-1]$. Such final subtractions are also necessary for other modular reduction algorithms in order to obtain the least non-negative residue. In our experiments, we were able to identify final subtractions in both the power and EM trace, which means that an attack like the one described in [25] is possible against our unprotected implementation.

Based on these experimental attacks, it is evident that the vulnerabilities of the unprotected implementation are mainly due to conditional statements, which can be easily identified in power as well as EM traces. These conditional statements (e.g. if-then constructs) are present in the square-and-multiply algorithm and Montgomery reduction [25]. Countermeasures against SPA attack are, in general, fairly simple; we give in the following two examples of low-cost countermeasures to increase the SPA-resistance of an RSA implementation.

1) One needs to avoid using conditional statements in the exponentiation algorithm. Both the suqare-and-multiply algorithm and the standard version of the m-ary method (as described in [10] or [11]), perform a multiplication only if the corresponding bit or digit of the exponent is non-zero, which leaks SPA-relevant information. On the other hand, our variant of the m-ary method (shown in Algorithm 2) executes always four squarings followed by a multiplication, regardless of whether a digit of the exponent is zero or not. Therefore, Algorithm 2 allows for efficient protection against SPA attacks[1].
2) In order to protect the Montgomery multiplication and squaring against SPA, all conditional subtractions of the modulus N have to be replaced by unconditional subtractions of a subtrahend that is either N or a byte-array containing zeros. To get unconditional source code, we extended the byte-array holding the modulus N with a second array holding only zeroes. Then, all conditional branches can be replaced by index calculations into this special byte array to subtract either the actual modulus N, or only zero. Of course, also this index calculation has to done in an SPA-resistant way.

After integration of these simple countermeasures, we were not able anymore to mount a successful SPA attack.

B. DPA Attack and Countermeasure

Differential Power Analysis (DPA) is a kind of side-channel attack that utilizes statistical analysis to extract information related to secret keys. To show the practicability of DPA, we mounted a Multiple Exponent Single Data (MESD) attack [18] against our m-ary exponentiation with SPA countermeasures as sketched in the previous subsection. The MESD attack is a DPA-style attack applicable to modular exponentiation. This attack requires the processor executing the implementation to exponentiate the same message M with n different exponents E, i.e. $E_0, E_1, \ldots, E_n$. Even though RSA is usually performed with a fixed secret exponent, the MESD attack is nonetheless relevant in a situation where the attacker possesses a device with the same power consumption characteristics as the device under attack. In this case, he can use the identical device to capture power traces for different exponents. The basic idea of the MESD attack is as follows: the attacker exponentiates a constant message with the unknown secret key and several guesses for a part of the key, which is in our case simply a 4-bit digit of E. After each guess of an exponent-digit, the power or EM traces for the secret exponent and the guessed exponent are subtracted from one another to get the DPA bias trace [18]. By inspecting the DPA bias trace, the attacker can identify the correct guesses of the exponent digit by digit.

In our attack experiment, we used 256-bit RSA executed on an ATmega128 controller with a frequency of 7.37 MHz as an example. We mounted the attack against the m-ary method with SPA countermeasures (Algorithm 2). Each digit of the exponent consists of 4 bits in our case. After the 16 powers of M have been computed and stored in a look-up table, the exponentiation in fact starts at e_{s-2}, the second-most significant 4-bit digit of E, and works downward. We try to guess the exponent digit by digit. Assume we have already got the first $(i-1)$ digits of E, i.e. the DPA trace will exhibit a zero bias in the leading $(i-1)$ digit positions. Figure 3 shows the differential traces for the correct guess and for one (due to space limits) of the incorrect guesses for a 4-bit digit of the

[1]Note that we do not consider safe-error fault attacks in this paper.

exponent, obtained from 100 power traces each. One can see that the correct guess is perfectly distinguishable by a leading zero part.

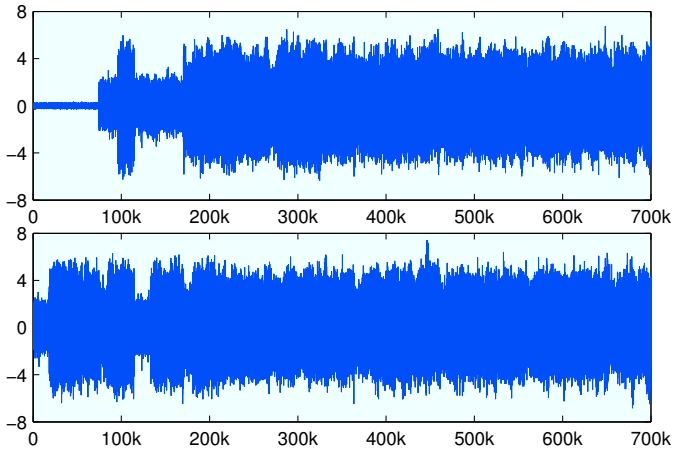

Fig. 3. MESD attack: differential traces for the correct guess (top) and one incorrect guess (bottom) for a 4-bit exponent digit.

From this experiment, we can see that our MESD attack is successful: we can recover the exponent digit by digit if the exponentiation is implemented according to the m-ary method (the attack worked for both the unprotected version and the version with SPA countermeasures). However, our RSA implementation can also resist DPA attacks after the integration of DPA countermeasures such as blinding [17]. We apply the message blinding method, which was originally introduced in [13]. First, we choose a random v_f co-prime to the modulus N and then obtain $v_i = (v_f{}^{-1})^E \bmod N$, where E denotes the public exponent. Before performing the exponentiation, the input message is multiplied by $v_i \bmod N$, and afterwards the result is corrected by multiplying it by $v_f \bmod N$. Inversion is, in general, a costly operation; therefore, it makes sense to pre-compute a (v_f, v_i) pair and square both v_f and v_i before each new exponentiation, as suggested in [13]. If (v_i, v_f) is kept secret, the attacker has no useful information about the input to the modular exponentiation.

In our case, just blinding the message is not enough since in RSA the same exponent is used several times, and hence exponent blinding (also introduced in [13]) is necessary. The basic idea is to "randomize" the exponent E by addition of a random multiple of $\phi(N) = (p-1)(q-1)$ before performing an exponentiation so that the bits e_i of E are different every time. A typical size of the random number by which $\phi(N)$ is multiplied is 32 bits. Putting it all together, an exponentiation with both message and exponent blinding is performed in the following way:

1) Blind the message M using v_i: $M' = v_i \cdot M \bmod N$.
2) Blind the exponent E using a random number r (for a 1024-bit RSA, r is typically 32 bits): $E' = E + r \cdot \phi N$.
3) Do exponentiation after blinding: $C' = M'^{E'} \bmod N$.
4) After receiving C', un-blind it to get the original value of C: $C = v_f \cdot C' \bmod N$.

In summary, the SPA and DPA countermeasures increase the execution time of the exponentiation by about 12% compared to an unprotected implementation.

V. Conclusions

The current Internet is, to a large extent, secured by RSA-based PKI, CAs, and security protocols. Expanding the scope of these well-established security protocols and services to the "Internet of Things" requires efficient RSA implementations for resource-constrained devices such as sensor nodes. In this paper we introduced a highly-optimized RSA implementation for 8-bit AVR microcontrollers that is able to perform a full 1024-bit private-key operation in less than $76 \cdot 10^6$ cycles on the ATmega128. We achieved this performance, which sets a new speed record for 1024-bit RSA on 8-bit platforms, thanks to an optimized variant of the hybrid multiplication technique that saves approximately 7.8% in execution time compared to the original hybrid method of Gura et al. To further improve performance, we integrated our new hybrid method into three different algorithms for Montgomery multiplication (namely SOS, FIPS, and KCM) and identified the hybrid SOS (HSOS) technique as the best option. Our implementation also makes use of the CRT and m-ary exponentiation method to achieve peak performance for private-key operations. In addition, we aimed to protect our implementation against power-analysis attacks, which is necessary in the "Internet of Things" since an attacker may have access to the communication endpoints so that he can monitor side-channel leakage. The countermeasures we integrated into our RSA implementation increase the execution time by roughly 12%, yet our protected private-key operation is faster than most unprotected implementations in the literature. In summary, our RSA implementation satisfies all requirements to expand the security protocols and services of the current Internet to the "Internet of Things"

Acknowledgments

We are grateful to to Jean-Sébastien Coron for his insights and helpful comments on various aspects of this work.

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, Jul. 2008.

[2] ——, "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, Jun. 2008.

[3] I. F. Blake, G. Seroussi, and N. P. Smart, *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[4] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, Dec. 1990.

[5] J. Deng, R. Han, and S. Mishra, "A performance evaluation of intrusion-tolerant routing in wireless sensor networks," in *Information Processing in Sensor Networks — IPSN 2003*, ser. Lecture Notes in Computer Science, vol. 2634. Springer Verlag, 2003, pp. 349–364.

[6] J. Großschädl, R. M. Avanzi, E. Savaş, and S. Tillich, "Energy-efficient software implementation of long integer modular arithmetic," in *Cryptographic Hardware and Embedded Systems — CHES 2005*, ser. Lecture Notes in Computer Science, vol. 3659. Springer Verlag, 2005, pp. 75–90.

[7] N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," in *Cryptographic Hardware and Embedded Systems — CHES 2004*, ser. Lecture Notes in Computer Science, vol. 3156. Springer Verlag, 2004, pp. 119–132.

[8] M. Joye and M. Tunstall, "Exponent recoding and regular exponentiation algorithms," in *Progress in Cryptology — AFRICACRYPT 2009*, ser. Lecture Notes in Computer Science, vol. 5580. Springer Verlag, 2009, pp. 334–349.

[9] A. A. Karatsuba and Y. P. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics - Doklady*, vol. 7, no. 7, pp. 595–596, Jan. 1963.

[10] D. E. Knuth, *Seminumerical Algorithms*, 3rd ed., ser. The Art of Computer Programming. Addison-Wesley, 1998, vol. 2.

[11] Ç. K. Koç, "High-speed RSA implementation," RSA Laboratories, RSA Data Security, Inc., Redwood City, CA, USA, Tech. Rep. TR 201, Nov. 1994, available for download at ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf.

[12] Ç. K. Koç, T. Acar, and B. S. Kaliski, "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.

[13] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology — CRYPTO '96*, ser. Lecture Notes in Computer Science, vol. 1109. Springer Verlag, 1996, pp. 104–113.

[14] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology — CRYPTO '99*, ser. Lecture Notes in Computer Science, vol. 1666. Springer Verlag, 1999, pp. 388–397.

[15] C. Lederer, R. Mader, M. Koschuch, J. Großschädl, A. Szekely, and S. Tillich, "Energy-efficient implementation of ECDH key exchange for wireless sensor networks," in *Information Security Theory and Practice — WISTP 2009*, ser. Lecture Notes in Computer Science, vol. 5746. Springer Verlag, 2009, pp. 112–127.

[16] 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)*. IEEE Computer Society Press, 2008, pp. 245–256.

[17] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Verlag, 2007.

[18] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Power analysis attacks of modular exponentiation in smartcards," in *Cryptographic Hardware and Embedded Systems — CHES '99*, ser. Lecture Notes in Computer Science, vol. 1717. Springer Verlag, 1999, pp. 144–157.

[19] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[20] J.-J. Quisquater and C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystem," *Electronics Letters*, vol. 18, no. 21, pp. 905–907, Oct. 1982.

[21] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[22] M. Scott and P. Szczechowiak, "Optimizing multiprecision multiplication for public key cryptography," Cryptology ePrint Archive, Report 2007/299, 2007, avialable for download at http://eprint.iacr.org.

[23] L. Uhsadel, A. Poschmann, and C. Paar, "Enabling full-size public-key algorithms on 8-bit sensor nodes," in *Security and Privacy in Ad-hoc and Sensor Networks — SASN 2007*, ser. Lecture Notes in Computer Science, vol. 4572. Springer Verlag, 2007, pp. 73–86.

[24] VeriSign, Inc., "Secure Wireless E-Commerce with PKI from VeriSign," White paper, available for download at https://www.verisign.com/server/rsc/wp/wap/index.html, Jan. 2000.

[25] C. D. Walter and S. Thompson, "Distinguishing exponent digits by observing modular subtractions," in *Topics in Cryptology — CT-RSA 2001*, ser. Lecture Notes in Computer Science, vol. 2020. Springer Verlag, 2001, pp. 192–207.

[26] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. Chang Shantz, "Energy analysis of public-key cryptography for wireless sensor networks," in *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communication (PerCom 2005)*. IEEE Computer Society Press, 2005, pp. 324–328.

[27] H. Wang and Q. Li, "Efficient implementation of public key cryptosystems on mote sensors," in *Information and Communications Security — ICICS 2006*, ser. Lecture Notes in Computer Science, vol. 4307. Springer Verlag, 2006, pp. 519–528.

[28] R. Watro, D. Kong, S.-F. Cuti, C. Gardiner, C. Lynn, and P. Kruus, "TinyPK: securing sensor networks with public key technology," in *Proceedings of the 2nd ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN 2004)*. ACM Press, 2004, pp. 59–64.