Reverse Product-Scanning Multiplication and Squaring on 8-Bit AVR Processors

Zhe Liu[1(✉)], Hwajeong Seo[2], Johann Großschädl[1], and Howon Kim[2]

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

[2] School of Computer Science and Engineering, Pusan National University, San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{hwajeong,howonkim}@pusan.ac.kr

Abstract. High performance, small code size, and good scalability are important requirements for software implementations of multi-precision arithmetic algorithms to fit resource-limited embedded systems. In this paper, we describe optimization techniques to speed up multi-precision multiplication and squaring on the AVR ATmega series of 8-bit microcontrollers. First, we present a new approach to perform multi-precision multiplication, called *Reverse Product Scanning (RPS)*, that resembles the hybrid technique of Gura et al., but calculates the byte-products in the inner loop in reverse order. The RPS method processes four bytes of the two operands in each iteration of the inner loop and employs two carry-catcher registers to minimize the number of `add` instructions. We also describe an optimized algorithm for multi-precision squaring based on the RPS technique that is, depending on the operand length, up to 44.3 % faster than multiplication. Our AVR Assembly implementations of RPS multiplication and RPS squaring occupy less than 1 kB of code space each and are written in a parameterized fashion so that they can support operands of varying length without recompilation. Despite this high level of flexibility, our RPS multiplication outperforms the looped variant of Hutter et al.'s operand-caching technique and saves between 40 and 51 % of code size. We also combine our RPS multiplication and squaring routines with Karatsuba's method to further reduce execution time. When executed on an ATmega128 processor, the "karatsubarized RPS method" needs only 85 k clock cycles for a 1024-bit multiplication (or 48 k cycles for a squaring). These results show that it is possible to achieve high performance without sacrificing code size or scalability.

1 Introduction

Multi-precision multiplication and squaring are performance-critical operations of a variety of public-key cryptographic algorithms, including RSA [18], elliptic curve schemes [14,17], and pairing-based cryptosystems [4]. In fact, these two operations can easily account for more than 80 % of the overall execution time

L.C.K. Hui et al. (Eds.): ICICS 2014, LNCS 8958, pp. 158–175, 2015.
DOI: 10.1007/978-3-319-21966-0_12

of a modular exponentiation (such as needed for RSA) or scalar multiplication (needed in elliptic curve cryptography [10]). Consequently, any effort spent on optimizing multi-precision multiplication and squaring is well spent. This is in particular the case for multi-precision arithmetic to be executed on embedded or mobile devices as they are often severely restricted in processing power and memory (RAM) capacity. For example, an ordinary smart card or sensor node features just an 8-bit processor clocked with a frequency of between 5 and 10 MHz. The 8-bit AVR architecture is widely used in these application domains and has, therefore, been the target platform numerous research projects in the area of lightweight implementation of cryptographic primitives. A typical 8-bit AVR processor (e.g. ATmega128 [2]) has 4 kB RAM, 128 kB flash memory to store program code, and provides 32 registers. Three register pairs can serve as 16-bit pointer registers and hold the address of operands in RAM [1]. The ATmega128 also comes with a hardware multiplier that needs two clock cycles to compute the 16-bit product of two 8-bit operands held in registers.

In the past ten years, a large body of research has been devoted to improve the performance of multi-precision arithmetic operations on resource-restricted 8-bit platforms such as the ATmega family of processors. At CHES 2004, Gura et al. presented a landmark paper in which they compared ECC with RSA on 8-bit CPUs and introduced the now-classical *hybrid method* for multiplication [9]. The hybrid method exploits the large register file of the AVR platform to store several bytes of the operands in registers and, in this way, combines the advantages of the *product scanning* and *operand scanning* technique [10]. Since the publication of Gura et al.'s work, there have been a number of attempts to further improve hybrid multiplication. An obvious approach for optimization is to completely unroll the loops since loop unrolling eliminates a lot of overhead (e.g. update of a loop counter or execution of a branch instruction) and allows for a specific "tuning" of each iteration (see e.g. [19,23]), which is not possible with "rolled" loops. A second line of research focused on speeding up the inner-loop operation, i.e. Multiply-ACcumulate (MAC) operation, by scheduling the execution of `mul` instructions in a special way and other low-level optimization techniques (see e.g. [15,16,24] for representative examples).

A recent milestone in the area of fast multi-precision multiplication on the 8-bit AVR platform is the *operand caching method*, introduced by Hutter and Wenger at CHES 2011 [12]. The operand caching method follows a similar idea as hybrid multiplication, but splits the computation of the product into several small(er) parts with the goal of reducing the overall number of `ld` instructions through a sophisticated caching of operand bytes. Later, fully unrolled versions of the operand caching approach with slightly better results were presented in [20,21] by Seo et al. Very recently, Hutter and Schwabe [11] further improved the speed record for unrolled multi-precision multiplication on AVR processors by a carefully-optimized implementation of Karatsuba's multiplication method [13]. Their results demonstrate that the operand length at which Karatsuba's approach starts to become beneficial is surprisingly low, namely 48 bits.

In this paper, we describe a new approach for multi-precision multiplication and squaring on 8-bit AVR processors and other platforms that feature a large number of general-purpose working registers. Our main contribution is the *Reverse Product Scanning (RPS) method* for multiplication, which resembles the basic loop structure of Gura et al.'s hybrid technique [9], but computes subsets of the partial products in the inner loop in reverse order, i.e. from more to less significant positions. Furthermore, the RPS method uses two so-called "carry-catcher" registers to minimize the number of `adc` instructions executed in the inner loop. The RPS method for AVR processors we present in this paper aims for a practical trade-off between performance, code size and scalability instead of "pure speed" as most other implementations. Achieving such a trade-off is a very challenging task since, for example, high performance and small code size usually contradict each other. Before presenting our contributions in detail, we first explain why fast execution time, small code size, and high scalability are all important requirements for multi-precision arithmetic software.

Requirements

Achieving fast execution time has been a major goal of virtually all implementations of multi-precision arithmetic described in the recent literature, and the present work is no exception. As stated before, the efficiency of multi-precision multiplication and squaring has a clear and direct impact on the performance of higher-level operations like exponentiation or scalar multiplication, which, in turn, determines the overall processing time of key establishment mechanisms and signature schemes. Besides reducing the delay of public-key primitives and protocols, fast multi-precision arithmetic is crucial for another reason, namely energy efficiency. In general, the energy consumption of cryptographic software executed on a microprocessor increases proportionally with the execution time [7]. This, together with the fact that a large portion of embedded systems are battery powered, makes a good case for minimizing execution time, even if the target application does not impose stringent delay constraints on cryptographic primitives. While the importance of high performance is unanimously accepted in the cryptographic community, the situation is not so clear when it comes to code size and scalability since both were often ignored in previous work.

Paying attention to code size is important since the binary executable image of an application (which includes the object file containing the arithmetic functions) needs to fit into the available program ROM or flash memory. Many embedded microcontrollers are quite restricted in code space; for example, the Atmel ATmega128 [2] provides only 128 kB of programmable flash memory to store program code. In light of such constraints, it is often unattractive (and in some cases even impossible) to apply certain code-size-increasing optimization techniques like (full) loop unrolling. The operand-caching method for AVR, as described in [12], serves as a good example to make this more clear. A fully unrolled implementation of operand-caching multiplication for 1024-bit operands has a binary code size of about 150 kB, which exceeds the flash capacity of the ATmega128 by more than 20 kB. But even for smaller operands typically used

in ECC (e.g. 256 bits), full loop unrolling can be infeasible since, depending on the code size of the operating system, networking stack, security protocol, and the actual application, only a tiny fraction of the 128 kB flash memory may be available for multiple-precision arithmetic. And, of course, the smaller the code size of the cryptographic primitives and underlying arithmetic operations, the more code space remains for the actual application.

In the context of cryptographic software, the term scalability relates to the ability to process operands of arbitrary size without the need to re-write or re-compile the software. A scalable implementation of multi-precision arithmetic is parameterized, which means that the operands (or, more precisely, pointers to the operands in RAM) are passed as parameters to the arithmetic function along with an additional parameter specifying the length of the operands. The function body executes the arithmetic operation in a "looped" fashion, whereby the operand length determines the number of loop iterations. Virtually all cryptographic libraries of practical relevance contain a scalable implementation of multi-precision arithmetic, and also the multiplication/squaring routines we describe in the following sections are scalable. The importance of scalability is best explained by taking RSA [18] as example. Private-key operations, such as decryption and signature generation, can exploit the Chinese Remainder Theorem to perform an n-bit exponentiation through two $(n/2)$-bit exponentiations using the prime decomposition P, Q of the modulus N. On the other hand, all operations involving a public key (e.g. encryption, signature verification) have to be performed on full-length (i.e. n-bit) operands. A scalable implementation of multiple-precision modular arithmetic is able to accommodate both operand lengths, which simplifies the software development process as only one function for multiplication and modular reduction has to be written. Scalability makes it also very easy to adapt an RSA implementation to larger key sizes, e.g. from 1024 to 1536 or 2048 bits.

Contributions

We present the RPS method for multi-precision multiplication and squaring on processors featuring a large number of general-purpose registers. As mentioned before, our RPS approach has the same loop structure as the hybrid technique and, consequently, employs the column-wise strategy (i.e. product scanning) as "outer algorithm." However, the byte-level multiplications in the inner loop are (partly) executed in reverse order, i.e. some more-significant byte-products are calculated before less-significant ones. Furthermore, the RPS method uses two so-called carry-catcher registers to reduce the propagation of carries (which, in turn, reduces the number of `adc` instructions), similar to the optimized hybrid method of Scott and Szczechowiak [19]. We also describe how to apply the RPS approach for multi-precision squaring and introduce an optimized technique to calculate the byte-level squares in the "main diagonal" that appear only once in the final result. Last but not least, we combine our RPS multiplication and squaring with Karatsuba's algorithm [13] to further reduce the execution time for large operands (i.e. ≥ 512 bits) such as used in RSA.

Table 1. Comparison of AVR implementations of multi-precision multiplication and squaring with respect to code size, scalability, and whether the implementation was evaluated in the original paper for operand lengths used in ECC or RSA (the letters U, L, P indicate whether an implementation is unrolled, looped, or parameterized)

Implementation	Code size	Scalable	RSA	ECC	Speed record
Multi-precision multiplication on 8-bit AVR processors:					
Gura et al. [9]	n/a		✓	✓	
Hutter et al. [11]	3.1–7.6 kB			✓	✓ (U)
Hutter et al. (L) [12]	1.5–1.9 kB		✓	✓	
Hutter et al. (U) [12]	3.7–151 kB			✓	
Liu et al. [16]	n/a		✓		
Scott et al. [19]	n/a			✓	
Seo et al. [21]	3.6–10 kB			✓	
Uhsadel et al. [23]	n/a			✓	
Zhang et al. [24]	n/a	✓		✓	
This work (RPS mul)	914 B	✓	✓	✓	✓ (L,P)
Multi-precision squaring on 8-bit AVR processors:					
Liu et al. [16]	1.5 kB		✓		
Seo et al. [22]	3.2–9.1 kB			✓	✓ (U)
This work (RPS sqr)	844 B	✓	✓	✓	✓ (L,P)

We implemented the proposed RPS multiplication and RPS squaring in Assembly language, and "karatsubarized" versions thereof in portable C using the Assembly functions as sub-routines. Our prototype implementations satisfy the requirements mentioned before; in particular, they are scalable and, thus, able to support operands of (essentially) arbitrary length. Both RPS multiplication and RPS squaring have a code size of less than 1 kB, which is small compared to other implementations described in the literature (see Table 1). Despite its scalability and compact size, our RPS multiplication for AVR is faster than the bulk of previous work, beaten only by the three fully unrolled implementations [11,12,21]. Most notably, the RPS technique we propose outperforms the looped variant[1] of the operand caching approach (see Sect. 4 for details).

2 Multiplication Techniques

In this section, we give a brief overview of standard algorithms and techniques for fast execution of multi-precision multiplication on a w-bit general-purpose processor. We assume that A and B are n-bit operands represented by arrays

[1] A looped implementation has "rolled" loops [12], but in contrast to a parameterized implementation, the number of iterations is "hard-coded" and, hence, fixed. Looped implementations are smaller, but also slower, than their unrolled counterparts.

of $s = \lceil n/w \rceil$ single-precision (i.e. w-bit) words a_i and b_i, which means we have $A = (a_{s-1}, \ldots, a_0)$ and $B = (b_{s-1}, \ldots, b_0)$ with $0 \leq a_i, b_i < 2^w$ for $0 \leq i < s$.

2.1 Operand Scanning Method

A simple and easy-to-implement technique for multi-precision multiplication is the operand scanning method [10], also known as schoolbook method [7]. The operand scanning method, as specified in [7, Algorithm 1], has a characteristic nested-loop structure with an outer loop iterates through the s words b_i of the operand B, starting with the least-significant word b_0. In the inner loop, b_i is multiplied with a word a_j of operand A and the $2w$-bit product is added to the intermediate result obtained so far. More precisely, the operation performed in the inner loop is a special Multiply-Accumulate (MAC) operation of the form $(u, v) \leftarrow a \cdot b + c + d$, whereby (u, v) represents a double-precision (i.e. $2w$-bit) quantity and a, b, c, and d are all single-precision words.

When implemented for an 8-bit ATmega128 processor, the MAC operation consists of a `mul`, two `add`, and two `adc` (i.e. add-with-carry) instructions. The operand scanning method is fairly easy to program in a high-level language like C or Java [7], but is generally less efficient than the product scanning method (described below) if both are written in Assembly language. In summary, when multiplying two s-word operands, the operand scanning method has to execute s^2 `mul`, $4s^2$ `add` (resp. `adc`), $2s^2 + s$ `ld` (i.e. load), as well as $s^2 + s$ `st` (store) instructions [7].

2.2 Product Scanning Method

An alternative way of performing multi-precision multiplication is the product scanning method, sometimes credited to Paul Comba [6], who was the first to describe an efficient implementation of this method on an Intel processor. The product scanning method (specified in [7, Algorithm 2]) comprises two nested loops; one computes the lower half of the result and the second contributes the upper half. As the name suggests, the two outer loops of the product scanning method move through the product itself, starting at the least significant word [6,7]. More precisely, the product is obtained one word at a time, whereby the i-th word contains all partial products $a_j \cdot b_k$ with $j + k = i$. A graphical representation of the product scanning method (see e.g. [8, Fig.1]) shows that the partial products are processed in a column-wise fashion, whereas the operand scanning technique outlined above follows a row-wise schedule. The inner loop of the product scanning method executes a simple MAC operation of the form $(t, u, v) \leftarrow (t, u, v) + a \cdot b$, which means two w-bit words are multiplied and the $2w$-bit product is added to a cumulative sum held in three w-bit registers.

An AVR Assembly implementation of the inner-loop operation consists of a `mul`, an `add`, and two `adc` instructions. Hence, the product scanning technique executes one `add` instruction less in the inner loop than the operand scanning method. Furthermore, the product scanning technique performs memory-write (i.e. store) operations exclusively in the outer loop(s), which means the overall

number of `st` instructions grows linearly with s instead of quadratically. When multiplying two s-word operands, the product scanning method has to execute s^2 `mul`, $3s^2$ `add` (resp. `adc`), $2s^2$ `ld`, and $2s$ `st` instructions [7].

2.3 Hybrid Method

Both the operand scanning and the product scanning method require (at least) $2s^2$ `ld` instructions if the two operands consist of s words. The hybrid method aims at reducing the number of `ld` instructions on processors with a large register file by processing $d \geq 2$ words of A and B at once in each iteration of the inner loop. From an algorithmic point of view, the hybrid method combines the two techniques described in the previous subsections, which means it employs the product scanning approach as "outer algorithm" and the operand scanning approach as "inner algorithm" [9]. In each iteration of the inner loops, d words of A and d words of B are loaded from memory, multiplied together and added to a cumulative sum held in $2d + 1$ general-purpose registers. By doing so, the number of loop iterations and, hence, the number of `ld` instructions is reduced by a factor of d. The speed-up achievable through the hybrid method depends on d, which, in turn, is determined by the number of available registers.

Most hybrid implementations for AVR processors with 32 working registers use $d = 4$, which means the hybrid method has to execute just a quarter of the `ld` instructions of the straightforward product scanning method. However, this saving usually comes at the expense of an increased number of `add` (resp. `adc`) or `mov` (resp. `movw`) instructions.

2.4 Operand Caching Method

The operand caching technique, introduced in [12], is currently the fastest quadratic-complexity multiplication method for 8-bit ATmega processors. Thanks to a sophisticated caching of operand bytes, it held the speed record for looped multi-precision multiplication until now. The operand caching method follows the basic approach of product scanning, but divides the calculation into several row sections. By reordering the execution of inner and outer row sections, the operand caching method can reuse the operands that have already been loaded into working registers to generate the next partial product(s). In this way, the overall number of memory access operations, in particular `ld` instructions, can be massively reduced. The actual performance of the operand-caching method depends on the size of a row, i.e. the number of words of one operand that can be kept (i.e. "cached") in working registers. In total, the operand caching method performs $3s^2/e + s$ memory access operations, whereby e denotes the row size [12]. Among these memory accesses are $2s^2/e$ loads and $s^2/e + s$ stores.

On an ATmega processor, e can be as high as 10 when the operand caching method is implemented in a completely unrolled fashion, or 9 in the case of a looped implementation. Since $e \gg d$, the operand caching method outperforms the hybrid method by approximately 15 % on average [12].

2.5 Karatsuba Multiplication

The most important multiplication method with sub-quadratic complexity was introduced by Karatsuba in the early 1960s [13]. Karatsuba's approach reduces a multiplication of two operands consisting of s words to three multiplications of $(s/2)$-word operands and a couple of additions. The half-size multiplications can be performed with any multiplication technique, including the conventional operand-scanning and product-scanning method. Alternatively, it is possible to apply Karatsuba's idea recursively until the operands consist of just one single word, in which case the asymptotic complexity becomes $\theta(s^{log_2(3)})$.

There exist two variants of Karatsuba's multiplication technique, namely an additive form and a subtractive form [11]. Both require the s-word operands to be split up into a lower half consisting of the $k = \lceil s/2 \rceil$ least significant words and an upper half comprising the $\lfloor s/2 \rfloor = s - k$ most significant words, i.e. we have $A = A_H \cdot 2^{kw} + A_L$ whereby $A_L = A \bmod 2^{kw}$ and $A_H = A \operatorname{div} 2^{kw}$. The additive variant of Karatsuba's method obtains the product $A \cdot B$ through the following equation.

$$A_H B_H \cdot 2^{2kw} + [(A_H + A_L)(B_H + B_L) - A_H B_H - A_L B_L] \cdot 2^{kw} + A_L B_L \quad (1)$$

On the other hand, the subtractive variant computes $A \cdot B$ as follows.

$$A_H B_H \cdot 2^{2kw} + [A_H B_H + A_L B_L - (A_H - A_L)(B_H - B_L)] \cdot 2^{kw} + A_L B_L \quad (2)$$

Consequently, Karatsuba's method performs a multiplication of size s via three multiplications and eight additions of size $s/2$ (plus a potential carry propagation). In 2009, Bernstein [3] refined Karatsuba's technique to save an addition of size $s/2$, which slightly improves performance. Hutter and Schwabe describe in [11] a carefully optimized AVR implementation of the subtractive Karatsuba technique for operands of up to 256 bits that currently holds the speed record for multi-precision multiplication on an 8-bit processor.

3 Our Implementation

In the next two subsections, we describe the RPS technique for multi-precision multiplication and squaring on 8-bit AVR processors in full detail, whereby we first present the "big picture" (i.e. the algorithm itself) and then concentrate on the inner-loop operation.

3.1 Loop Structure

From an algorithmic point of view, our RPS multiplication method is similar to Gura et al.'s hybrid technique [9] since both resemble the basic loop structure of the classical product-scanning approach [10]. Consequently, the RPS method computes the product $A \cdot B$ through two nested loops one word at a time. The first nested loop produces the s least significant words, while the second nested

Algorithm 1. Multiple-precision multiplication

Input: Two s-word operands $A = (A_{s-1}, \ldots, A_1, A_0)$ and $B = (B_{s-1}, \ldots, B_1, B_0)$
Output: $2s$-word product $R = A \times B = (R_{2s-1}, \ldots, R_1, R_0)$

```
 1: Z ← A_0 × B_0
 2: R_0 ← Z mod 2^w ; Z ← Z/2^w
 3: for i from 1 by 1 to s − 1 do
 4:     k ← i + 1
 5:     for j from 0 by 1 to i do
 6:         k ← k − 1
 7:         Z ← Z + A_j × B_k
 8:     end for
 9:     R_i ← Z mod 2^w ; Z ← Z/2^w
10: end for
11: for i from s by 1 to 2s − 3 do
12:     k ← s
13:     for j from i − (s − 1) by 1 to s − 1 do
14:         k ← k − 1
15:         Z ← Z + A_j × B_k
16:     end for
17:     R_i ← Z mod 2^w ; Z ← Z/2^w
18: end for
19: Z ← Z + A_{s-1} × B_{s-1}
20: R_{2s-2} ← Z mod 2^w ; Z ← Z/2^w
21: R_{2s-1} ← Z mod 2^w
22: return (R_{2s-1}, ..., R_1, R_0)
```

loop yields the upper half of the $2s$-word product. When following the original product-scanning approach (as described in e.g. [7]), the bitlength w of a word is normally chosen to match the native word-size of the processor, which means $w = 8$ in the case of AVR, i.e. each word consists of a byte. However, since we process $d = 4$ bytes at a time, similar to the hybrid technique described in the previous section, our word-size is $w = 32$ (i.e. four bytes) despite the fact that we work on an 8-bit processor. To distinguish between words and bytes, we use from now on indexed capital letters to represent words, and indexed lowercase letters to denote the individual bytes a word is composed of. Consequently, an n-bit integer A consists of $s = \lceil n/32 \rceil$ words $A_i \in [0, 2^{32} - 1]$, each of which, in turn, contains four bytes, i.e. $A_i = (a_{4i+3}, a_{4i+2}, a_{4i+1}, a_{4i})$ for $0 \leq i < s$.

Algorithm 1 specifies the RPS multiplication technique using said notation for the w-bit words. The algorithm has the characteristic nested-loop structure of the classical product-scanning method and computes the product $A \times B$ in a column-wise fashion, one word at a time [7]. In each iteration of one of the two inner loops, a conventional Multiply-ACcumulate (MAC) operation of the form $Z \leftarrow Z + A_j \times B_k$ is executed, i.e. a w-bit word A_j of operand A is multiplied by a w-bit word B_k of operand B and the $2w$-bit product $A_j \times B_k$ is added to a cumulative sum Z. In our case, both A_j and B_k contain four bytes, while the sum Z is nine bytes long. Operations of the form $R_i \leftarrow Z \bmod 2^w$ simply write

Algorithm 2. Multiple-precision squaring

Input: An s-word operand $A = (A_{s-1}, \ldots, A_1, A_0)$
Output: $2s$-word square $R = A^2 = (R_{2s-1}, \ldots, R_1, R_0)$
1: $Z \leftarrow 0$; $R_0 \leftarrow 0$
2: **for** i from 0 by 1 to $s-1$ **do**
3: $k \leftarrow i+1$
4: **for** j from 0 by 1 to $k-2$ **do**
5: $k \leftarrow k-1$
6: $Z \leftarrow Z + A_j \times A_k$
7: **end for**
8: $R_i \leftarrow Z \bmod 2^w$; $Z \leftarrow Z/2^w$
9: **end for**
10: **for** i from s by 1 to $2s-3$ **do**
11: $k \leftarrow s$
12: **for** j from $i-(s-1)$ by 1 to $k-2$ **do**
13: $k \leftarrow k-1$
14: $Z \leftarrow Z + A_j \times A_k$
15: **end for**
16: $R_i \leftarrow Z \bmod 2^w$; $Z \leftarrow Z/2^w$
17: **end for**
18: $R_{2s-2} \leftarrow Z \bmod 2^w$; $R_{2s-1} \leftarrow 0$
19: $Z \leftarrow 0$
20: **for** i from 0 by 1 to $s-1$ **do**
21: $Z \leftarrow Z + A_i \times A_i + 2(R_{2i+1} \cdot 2^w + R_{2i})$
22: $R_{2i} \leftarrow Z \bmod 2^w$; $Z \leftarrow Z/2^w$
23: $R_{2i+1} \leftarrow Z \bmod 2^w$; $Z \leftarrow Z/2^w$
24: **end for**
25: **return** $(R_{2s-1}, \ldots, R_1, R_0)$

the w least significant bits (i.e. the four least significant bytes if $d = 4$) of Z to the destination R_i. The divisions of Z by 2^w (e.g. in line 2, 9, 17, and 20) are nothing else than simple w-bit (i.e. 4-byte) right shifts of Z. A further common characteristic between our RPS technique and the product-scanning method is that the words A_j of A are loaded in ascending order, starting with the least-significant word A_0, whereas the words B_k of operand B are loaded in opposite order, i.e. from more to less significant words.

Algorithm 1 differs from the straightforward product-scanning approach as described in e.g. [7,10] in a few details. First, we "peeled off" the very first and the very last MAC operation (in which $A_0 \times B_0$ and $A_{s-1} \times B_{s-1}$ are formed) from the nested loops and execute them outside the loop body. In this way, we do not need to initialize the sum Z with 0 and can replace the very first MAC operation by a simple multiplication. Also the last MAC operation allows for a special optimization to reduce execution time. Namely, after the very last MAC operation, we can directly write the eight least significant bytes of S to the two result words R_{2s-2} and R_{2s-1} in one pass without shifting S. Finally, the two

loop counters j and k are updated such that one can take full advantage of the automatic pre-decrement and post-increment addressing modes of AVR.

Algorithm 2 shows our implementation of multiple-precision squaring based on the product-scanning method. It is well known that the square $R = A^2$ of a long integer A can be computed much more efficiently than the product of two distinct integers due to the "symmetry" of partial products [10]. Namely, when a normal multiplication algorithm is used for squaring (e.g. Algorithm 1 if we set $B = A$), then all partial products of the form $A_j \times A_k$ with $j \neq k$ are computed twice because $A_j \times A_k = A_k \times A_j$. Dedicated squaring algorithms avoid such unnecessary overheads by calculating these partial products only once and then doubling them through a left-shift. Also the partial products in the "main diagonal" (i.e. the partial products of the form $A_i \times A_i$) appear exactly once in the final result and have to be treated separately. Algorithm 2 is based on this approach; it first computes the partial products $A_j \times A_k$ with $j \neq k$ and sums them up in a similar way as in product-scanning multiplication. Thereafter, the result obtained so far is doubled and the main diagonal containing the partial products of the form $A_i \times A_i$ is added.

The two nested loops of Algorithm 2 (i.e. line 2 to 17) compute the partial products $A_j \times A_k$ to be doubled and have a very similar structure as the loops of Algorithm 1. In fact, there are only two minor differences, namely that the very first and the last partial product are not peeled off from the nested loops anymore (since they form now part of the third loop) and that the inner loops are iterated fewer times. For example, the first inner loop (starting at line 4) is iterated while the condition $j \leq k - 2$ is true; in C-like programming languages this for-loop would be written as follows.

$$\texttt{for (j = 0; j <= k - 2; j++)}$$

As j is incremented and k decremented in each iteration of the inner loop, the overall number of loop iterations in roughly halved compared to the inner loop of the RPS multiplication in Algorithm 1. This is also the case with the second inner loop starting at line 12. Because of these modifications of the loop-termination conditions, the total number of $(w \times w)$-bit multiplications (resp. MAC operations) performed by the two loops is reduced from $s^2 - 2$ (Algorithm 1) to $(s^2 - s)/2$. In the third loop (line 20), the intermediate result produced by the two nested loops is doubled and the s partial products of the form $A_i \times A_i$ are added. Putting all three loops together, Algorithm 2 executes $(s^2 + s)/2$ MAC operations to obtain the square of an s-word integer, which is almost 50 % less compared to the s^2 MAC operations (or multiplications) of Algorithm 1.

3.2 Inner-Loop Operation

The two nested loops of both Algorithms 1 and 2 execute basic MAC operations of the form $Z \leftarrow Z + A_j \times B_k$ in their inner loops. As mentioned in the previous subsection, the two w-bit words A_j and B_k consist of $d = 4$ bytes each. Consequently, in each iteration of the inner loop(s), four bytes of operand A

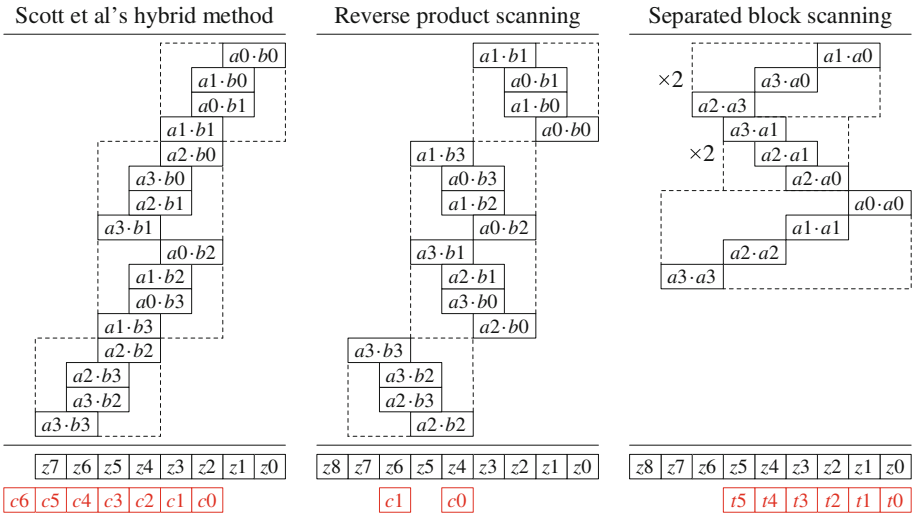

Fig. 1. Inner-loop operation based on Scott et al.'s carry-catcher method (left, taken from [19, Fig.1(ii)]), our RPS method using two carry-catcher registers (middle), and our SBS technique for computing the square of a 4-byte word (right)

are multiplied by four bytes of operand B and the 8-byte product is added to a cumulative sum Z consisting of nine bytes. The (4×4)-byte multiplication can be carried out in various different ways; for example, Gura et al. used the operand-scanning technique in their seminal paper [9]. Alternatively, it is also possible to apply the product-scanning method; Scott et al.'s implementation from [19] (depicted on the left side of Fig. 1) serves as a good example for this approach. If $d = 4$, a total of 16 byte-products needs to be computed, which is done from top to bottom, i.e. $a_0 \cdot b_0$ is generated first and $a_3 \cdot b_3$ is the last one to be processed. A particular issue when using the product-scanning technique in the inner loop is the propagation of carries; for example, the addition of the byte-product $a_0 \cdot b_0$ to the two least significant bytes (z_1, z_0) of the cumulative sum Z can produce a carry, which, in the worst case, may propagate up to the most significant byte of Z. To limit such carry propagation, Scott et al. introduced so-called *carry-catcher registers*, shown in red in Fig. 1. For $d = 4$, there are eight registers for the cumulative sum Z (which we denote as accu registers z_0 to z_7) and seven carry-catcher registers (c_0 to c_6). A carry generated by the addition of e.g. $a_0 \cdot b_0$ to (z_1, z_0) is not propagated along the z_i registers (up to z_7 in the worst case), but simply added to c_0. In this way, only an `add` and two `adc` instructions need to be performed to accumulate a byte-product. After the last iteration of the inner loop, the six carry-catcher registers are added to the accu registers to yield the correct Z, and then they are cleared.

Our approach of implementing the inner-loop operation is also based on the product-scanning technique and depicted in the middle of Fig. 1. In contrast to Scott et al., we have nine accu registers (z_0 to z_8), but only two carry catchers

(c_0 and c_1). To aid the explanation of our approach, we split the computation of the 16 byte-products up into four groups, indicated by four dashed boxes in Fig. 1. For example, the first group consists of $a_1 \cdot b_1$, $a_0 \cdot b_1$, $a_1 \cdot b_0$, as well as $a_0 \cdot b_0$. Compared to Scott et al., we compute the byte-products within a group in opposite order (see Fig. 1), and also the processing of the second and third group is reversed. The reversed-order computation of byte-products inspired us to call this approach *Reverse Product Scanning (RPS)*. Our main idea is to use the byte-products themselves to catch carries, which allows us to minimize the number of carry-catcher registers and speed up the computation.

The RPS method performs an iteration of the inner loop as follows. At the beginning, the four bytes b_3, b_2, b_1, and b_0 of a 32-bit word B_k are loaded from memory into four registers using the automatic pre-decrement addressing mode of the AVR architecture. Furthermore, we load the first two bytes of the word A_j, namely a_0 and a_1, taking advantage of post-increment addressing. Now, we multiply a_1 by b_1 and copy the 16-bit byte-product to two temporary registers t_0 and t_1 with help of the `movw` instruction. Register t_0 holds the "lower" byte of the product and t_1 the "upper" byte. Next, we form the product $a_0 \cdot b_1$ and add the product to accu register z_1 and the temporary register t_0. A potential carry from this addition can be safely added to the temporary register t_1 without overflowing it. Thereafter, we multiply a_1 by b_0, add the product $a_1 \cdot b_0$ to z_1 and t_0, and propagate the carry from the last addition to t_1. As before, it is not possible to overflow t_1, not even in the most extreme case where the bytes a_0, b_0, a_1, b_1, as well as the involved accu byte z_1, have the maximum possible of 255. After computation of the last byte-product of the first block (which is $a_0 \cdot b_0$), we add it together with the content of the temporary registers t_0, t_1 to the four accu registers z_0, z_1, z_2, z_3, and, finally, propagate the carry bit from the last addition to the carry-catcher register c_0. Overall, the processing of the first dashed block in the middle of Fig. 1 takes four `mul`, one `movw`, and a total of 11 `add` or `adc` instructions, respectively.

The second and third block are processed in essentially the same way as the first one; the only difference is the loading of the remaining two operand bytes of A_j, i.e. a_2 and a_3. Again, we use a carry-catcher register, namely c_1, to deal with the carry that may be generated when adding the last byte-product along with the content of the temporary registers t_0 and t_1 to the four accu registers z_2, z_3, z_4, and z_5. The two operand bytes a_2 and a_3 are loaded right after the computation of the second block, since a_0 and a_1 are not needed anymore. We load a_2 into the register holding a_0 and a_3 into the register of a_1, thereby overwriting a_0 and a_1. In summary, the second and third block execute exactly the same number of instructions as the first block. The fourth block, in which the final four byte-products are generated and added to the accu registers, differs slightly from the former three because no carry-catcher register is needed. Once a_2 has been multiplied by b_2, we add the concatenation of $a_2 \cdot b_2$ with the temporary registers t_0, t_1 to the four accu registers z_4, z_5, z_6, and z_7. However, the carry from the last addition is directly propagated to the most significant accu

register z_8. Consequently, the fourth block of byte-products executes the same number of `add`/`adc` instructions as the first three blocks, namely 11.

Putting all blocks together, the MAC operation for $d = 4$ comprises a total of eight `ld` (i.e. load), four `movw`, 16 `mul`, and 44 `add` or `adc` instructions. On an ATmega128 processor, these instruction counts translate to an execution time of exactly 96 clock cycles [2]. The overall execution time of one iteration of the inner loop (including incrementation of a loop counter and branch instruction) amounts to 99 clock cycles.

The first two nested loops of our RPS squaring technique (Algorithm 2) are very similar to that of RPS multiplication (Algorithm 1), only the termination conditions of the inner loops differ. In particular, the MAC operation executed in the two inner loops is exactly the same and can be implemented in the same way as discussed before. The third loop (line 20 to 24 in Algorithm 2) is unique in the sense that it is only needed for squaring. It is a simple (i.e. "un-nested") loop and squares a w-bit (i.e. 4-byte) word A_i in each iteration. Furthermore, a $2w$-bit quantity of the form $R_{2i+1} \cdot 2^w + R_{2i}$ is doubled and then added to the $2w$-bit square $A_i \times A_i$. The $2w$-bit quantity $R_{2i+1} \cdot 2^w + R_{2i}$ is made up of two w-bit words that form part of the intermediate result of the first two loops; in our case it is simply a 64-bit word of which R_{2i+1} is the upper half and R_{2i} the lower half. Since we have $d = 4$, any w-bit (i.e. 4-byte) word A_i can be squared by computing $(d^2 + d)/2 = 10$ byte-products; six of these have to be doubled and the remaining four not. Therefore, it makes sense to split the computation of $A_i \times A_i$ up into blocks and separate the computation of the byte-products to be doubled from the ones that are not doubled. This technique, which we call *Separated Block Scanning (SBS)*, is illustrated on the right of Fig. 1.

When using the SBS approach for the third loop, the computation of byte-products for a square $A_i \cdot A_i$ is organized in three blocks, indicated by dashed boxes in Fig. 1. At the beginning of an iteration, the four bytes of word A_i and eight bytes of the intermediate result R are loaded from RAM into 12 registers labeled with a_0 to a_3 and z_0 to z_7, respectively. Next, the register z_8 is copied to carry-catcher register c_0 and then cleared. After multiplication of the bytes a_1 and a_0, the product $a_1 \cdot a_0$ is moved to the temporary registers t_0 and t_1. In the next step, the byte-product $a_3 \cdot a_0$ is computed and moved to t_2, t_3. Once $a_2 \cdot a_3$ has been produced, we add all three byte-products to the accu registers z_1 to z_6 and propagate the carry generated by the last addition up to z_8. The second block starts with the multiplication of a_3 by a_0 and a `movw` instruction to copy the upper byte of $a_3 \cdot a_0$ to t_1 and the lower byte to t_0. Thereafter, we multiply a_2 by a_1, add the byte-product to the accu register pair (z_4, z_3), and propagate the carry into t_1. Finally, the byte-product $a_2 \cdot a_0$ is computed and added along with the content of (t_1, t_0) to the four accu registers z_2 to z_5. The carry from the last addition is again propagated up to z_8, which concludes the second block. Now, the nine accu registers z_0 to z_8 are doubled by executing an `add` and eight `adc` instructions. The third block contains all the byte-products that are not doubled, namely $a_l \cdot a_l$ for $0 \leq l \leq 3$. First, we compute the byte-product $a_0 \cdot a_0$, add the carry-catcher c_0 to it, and move the result to the two

Table 2. Execution time (in clock cycles) and code size (in bytes) of different multi-precision multiplication and squaring implementations for operands ranging from 160 to 512 bits on an ATmega128 (the letters U, L, P indicate whether an implementation is unrolled, looped, or parameterized; results marked with * are estimated results)

Implementation	Metric	160 bit	192 bit	224 bit	256 bit	384 bit	512 bit
AVR implementations of multi-precision multiplication:							
Hutter et al. (U) [11]	Time	2030	2987	n/a	4961	n/a	n/a
	Size	3106	4492	n/a	7616	n/a	n/a
Hutter et al. (U) [12]	Time	2396	3470	4694	6124	13702	24318
	Size	3778	5436	7340	9558	21350	37884
Seo et al. (U) [21]	Time	2346	3437	n/a	6128*	n/a	24205*
	Size	3662	n/a	n/a	n/a	n/a	n/a
Hutter et al. (L) [12]	Time	2693	3861	5267	6871	15457	27503
	Size	1562	1866	1538	1766	1614	1544
Liu et al. (P) [15]	Time	2778	4004	5398	7000	15488	27304
	Size	940	940	940	940	940	940
RPS Mul. (P)	Time	2690	3831	5170	6707	14835	26131
	Size	918	918	918	918	918	918
AVR implementations of multi-precision squaring:							
Seo et al. (U) [22]	Time	1456	2014	n/a	n/a	n/a	n/a
	Size	3204	5678	n/a	n/a	n/a	n/a
Liu et al. (L) [16]	Time	2375	3270	4305	5480	11580	19920
	Size	n/a	n/a	n/a	1542	n/a	n/a
RPS Sqr. (P)	time	1795	2457	3218	4078	8508	14522
	size	844	844	844	844	844	844

temporary registers t_0, t_1. Thereafter, the byte-products $a_1 \cdot a_1$ and $a_2 \cdot a_2$ are moved to t_2, t_3 and t_4, t_5, respectively. Finally, we compute $a_3 \cdot a_3$ and add all four byte-products in one pass to the eight accu registers z_0 to z_7, whereby the carry from the last addition is propagated into z_8.

The operation performed in the body of the third loop has a total execution time of 116 clock cycles (excluding counter update and branch instruction).

4 Performance Evaluation and Comparison

In this section, we report implementation results of the proposed RPS method for multiple-precision multiplication and squaring, including execution time (in clock cycles) and code size (in bytes). All timings were obtained by simulation with AVR studio version 4.19 using the ATmega128 [2] as target device.

Table 2 summarizes our results for different operand lengths (ranging from 160 to 512 bits) and compares them with execution time and code size figures

of related work. The first three rows show the best previous results for unrolled implementations of multiplication [11,12,21], while the fourth and fifth row in Table 2 contain the fastest looped [12] and parameterized [15] version, respectively. Thereafter, the results of our parameterized RPS method are given. The main conclusion that can be drawn from these results is that the RPS method is slightly faster (and much smaller) than the looped variant of Hutter et al.'s operand caching method [12]. While the difference is merely three clock cycles for 160-bit operands, it increases to about 5 % when the operands are 512 bits long. Our RPS technique sets new records for "not-fully-unrolled" (i.e. looped or parameterized) implementations of multiple-precision multiplication, beaten only by the fully unrolled implementations [11,12,21] at the cost of very large code size. For example, the unrolled operand caching technique for 512 bits has a code size of roughly 37.9 kB, which is almost 30 % of the flash memory of the ATmega128 processor [2]. For comparison, our parameterized implementation of RPS multiplication and squaring occupies less than 1 kB in flash each. RPS squaring is between 33.3 % (160-bit operands) and 44.3 % (512 bits) faster than RPS multiplication. In accordance with common practice, the timings given in Table 2 do not include the function-call overhead and the push/pop of "callee-saved" registers to/from the stack. Both together amounts to 89 clock cycles in the case of RPS multiplication and 85 cycles for RPS squaring.

We combined our RPS multiplication/squaring with Karatsuba's algorithm to further improve performance. More precisely, we implemented a subtractive Karatsuba variant as described in [11, Sect.3] to get a regular execution profile and constant execution time. Part of this effort was to implement a "constant-time conditional negation," which we did as proposed in [11]. However, unlike [11], we decided to not merge all sub-operations of a Karatsuba multiplication

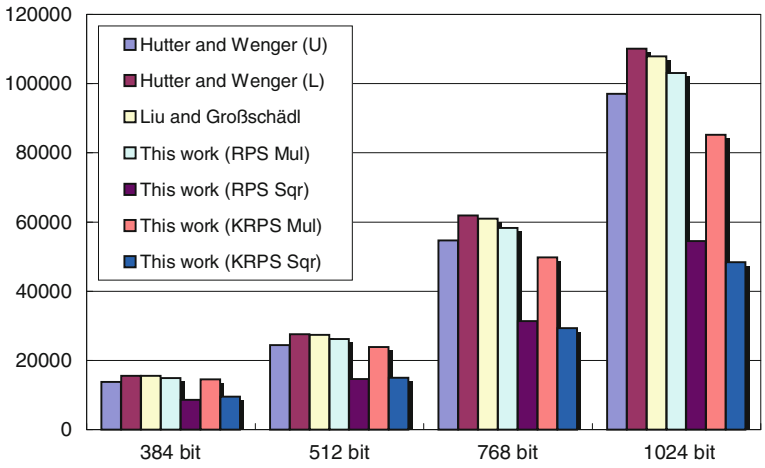

Fig. 2. Execution time (in clock cycles) of different implementations of multiplication and squaring for "large" operand sizes ranging from 384 to 1024 bits

into a single function, but execute them by calling low-level functions (such as subtraction, RPS multiplication, negation) to minimize code size. A graphical comparison of the execution times of our "karatsubarized" RPS multiplication (KRPS multiplication in short) and squaring (i.e. KRPS squaring) is shown in Fig. 2. Since the KRPS multiplication calls 12 low-level functions (which introduces a total function-call overhead of several 100 cycles), Karatsuba's method starts to become beneficial only for relatively large operands, namely 384 bits for multiplication and 640 bits in the case of squaring. When the operands have a length of 512 bits or more, our KRPS variant even outperforms the unrolled operand-caching approach. To give two concrete results, a KRPS multiplication needs 85 k cycles for 1024-bit operands, while a squaring takes 48 k cycles.

5 Conclusions

In this paper we advanced the state-of-the-art in multiple-precision multiplication and squaring on 8-bit AVR processors. We first presented some arguments in favor of parameterized implementations of multiple-precision arithmetic and pointed out that, besides execution time, also scalability and code size deserve consideration. Our main contribution is the RPS technique multiplication and squaring, which follows the basic approach of Gura et al.'s hybrid method from CHES 2004, but optimizes the execution of MAC operations in the inner loops by reversing the order of the byte multiplications. Experimental results show a clear advantage of our RPS technique over a looped realization of the operand caching approach; we are not only faster, but also smaller in terms of code size (e.g. 51 % for 192-bit operands). Combining our RPS method with Karatsuba's idea allowed us to achieve record-setting execution times for multiplication and squaring of operands of a length of 512 bits and beyond. In summary, our work shows that high performance does not necessarily have to come at the expense of poor scalability and/or large code size.

References

1. Atmel Corporation: 8-bit ARV® Instruction Set. User Guide, July 2008. http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf
2. Atmel Corporation: 8-bit ARV® Microcontroller with 128 K Bytes In-System Programmable Flash: ATmega128, ATmega128L. Datasheet, June 2008. http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf
3. Bernstein, D.J.: Batch binary Edwards. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 317–336. Springer, Heidelberg (2009)
4. Boneh, D., Franklin, M.K.: Identity-based encryption from the Weil pairing. SIAM J. Comput. **32**(3), 586–615 (2003)
5. Brent, R.P., Zimmermann, P.: Modern Computer Arithmetic, Cambridge Monographs on Applied and Computational Mathematics, vol. 18. Cambridge University Press, Cambridge (2010)
6. Comba, P.G.: Exponentiation cryptosystems on the IBM PC. IBM Syst. J. **29**(4), 526–538 (1990)

7. Großschädl, J., Avanzi, R.M., Savaş, E., Tillich, S.: Energy-efficient software implementation of long integer modular arithmetic. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 75–90. Springer, Heidelberg (2005)
8. Großschädl, J., Savaş, E.: Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 133–147. Springer, Heidelberg (2004)
9. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 119–132. Springer, Heidelberg (2004)
10. Hankerson, D.R., Menezes, A.J., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer Verlag, New York (2004)
11. Hutter, M., Schwabe, P.: Multiprecision multiplication on AVR revisited. Cryptology ePrint Archive, Report 2014/592 (2014). http://eprint.iacr.org/
12. Hutter, M., Wenger, E.: Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 459–474. Springer, Heidelberg (2011)
13. Karatsuba, A.A., Ofman, Y.P.: Multiplication of multidigit numbers on automata. Soviet Physics - Doklady **7**(7), 595–596 (1963)
14. Koblitz, N.I.: Elliptic curve cryptosystems. Math. Comput. **48**(177), 203–209 (1987)
15. Liu, Z., Großschädl, J.: New speed records for montgomery modular multiplication on 8-Bit AVR microcontrollers. In: Pointcheval, D., Vergnaud, D. (eds.) AFRICACRYPT. LNCS, vol. 8469, pp. 215–234. Springer, Heidelberg (2014)
16. Liu, Z., Großschädl, J., Kizhvatov, I.: 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) (2010)
17. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
18. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
19. Scott, M., Szczechowiak, P.: Optimizing multiprecision multiplication for public key cryptography. Cryptology ePrint Archive, Report 2007/299 (2007). http://eprint.iacr.org
20. Seo, H., Kim, H.: Multi-precision multiplication for public-key cryptography on embedded microprocessors. In: Lee, D.H., Yung, M. (eds.) WISA 2012. LNCS, vol. 7690, pp. 55–67. Springer, Heidelberg (2012)
21. Seo, H., Kim, H.: Optimized multi-precision multiplication for public-key cryptography on embedded microprocessors. Int. J. Comput. Commun. Eng. **2**(3), 255–259 (2013)
22. Seo, H., Liu, Z., Choi, J., Kim, H.: Multi-precision squaring for public-key cryptography on embedded microprocessors. In: Paul, G., Vaudenay, S. (eds.) INDOCRYPT 2013. LNCS, vol. 8250, pp. 227–243. Springer, Heidelberg (2013)
23. Uhsadel, L., Poschmann, A., Paar, C.: Enabling full-size public-key algorithms on 8-bit sensor nodes. In: Stajano, F., Meadows, C., Capkun, S., Moore, T. (eds.) ESAS 2007. LNCS, vol. 4572, pp. 73–86. Springer, Heidelberg (2007)
24. Zhang, Y., Großschädl, J.: 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), vol. 1, pp. 459–466. IEEE (2011)