Efficient Prime-Field Arithmetic for Elliptic Curve Cryptography on Wireless Sensor Nodes

Yang Zhang
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
L–1359 Luxembourg
yang.zhang.001@student.uni.lu

Johann Großschädl
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
L–1359 Luxembourg
johann.groszschaedl@uni.lu

ABSTRACT

Public-Key Cryptography (PKC) is essential to ensure the authenticity and confidentiality of communication in open computer networks such as the Internet. While RSA is still the most widely used public-key cryptosystem today, it can be expected that Elliptic Curve Cryptography (ECC) will continue to gain importance and become the de-facto standard for PKC in the emerging "Internet of Things." ECC is particularly attractive for use in resource-restricted devices (e.g. wireless sensor nodes, RFID tags) due to its high level of security per bit, which allows for shorter keys compared to RSA. The performance of elliptic curve cryptosystems is primarily determined by the efficiency of certain arithmetic operations (especially multiplication and squaring) in the underlying finite field. In the present paper, we introduce a high-speed implementation of arithmetic in Optimal Prime Fields (OPFs) for the ATmega128, an 8-bit processor used in a number of sensor nodes including the MICAz mote. An OPF is defined by a prime of the form $p = u \cdot 2^k + v$, whereby u and v are small compared to 2^k; in our implementation u is a 16-bit integer and $v = 1$. A special property of these primes is their low Hamming weight since only a few bits near the MSB and LSB are one. We describe an optimized variant of Montgomery multiplication, based on Gura et al's hybrid technique, that takes the low weight of such primes into account to minimize execution time. Our implementation for the ATmega128 is able to perform a multiplication in a 160-bit OPF in 3,542 clock cycles, which represents a new speed record for 160-bit modular multiplication on an 8-bit processor.

1. INTRODUCTION

A Wireless Sensor Network (WSN) can be broadly defined as a self-configuring network of autonomous sensing devices (called *motes*), which are deployed in an area of interest to cooperatively monitor a certain phenomenon or condition (e.g. temperature) [1]. WSNs are envisioned to provide the missing link between the physical world we live in and the digital world of computers. In the recent past, WSNs have attracted considerable attention and found widespread use in a multitude of applications ranging from environmental surveillance over medical monitoring to home automation and object tracking [20]. A typical sensor node, such as the MICAz mote [3], features an 8-bit processor clocked with a frequency of between 4 and 12 MHz, a few kB or RAM, a larger amount of ROM and/or flash memory, an RF module compliant to the IEEE 802.15.4 ("ZigBee") standard, two AA batteries, and one or more sensors. Consequently, the MICAz mote can be seen as a battery-powered miniature computer with sensing and wireless networking capabilities [1, 20]. However, WSNs differ from "conventional" networks that connect commodity computers (e.g. LANs) in various aspects; for example, WSNs are highly self-organized and fault-tolerant, their nodes have limited energy supply and hence restricted processing power, and the communication among the nodes is characterized by low transmission rates and multi-hop routing.

Security and privacy issues pose a great challenge for the current and future adoption of WSN technology in certain application domains such as health care, traffic control, and disaster detection [20]. Unfortunately, WSNs are easier to attack (and, hence, harder to protect) than a conventional network (e.g. an Ethernet LAN) since the sensor nodes are often deployed in unattended environments, which implies an attacker may be able to directly access individual nodes [1]. In this case, he can capture one or more nodes, perform all kinds of physical attacks to extract secret keys, manipulate the nodes, and then inject manipulated nodes into the network with the goal to compromise the correct operation and/or security of the WSN [15]. Therefore, WSNs require a sophisticated security architecture that takes these special threat scenarios (and adversary models) into account. Two crucial building blocks of virtually all security frameworks for WSNs described in scientific papers are authentication and key establishment [19]. In an "ordinary" computer network, the authentication of an entity (or user) as well as the establishment of a secret key shared between two entities can be effectively performed using public-key cryptosystems such as RSA, DSA, Diffie-Hellman, or their elliptic curve variants [9]. However, most security protocols designed on basis of computation-intensive public-key cryptography are not straightforwardly adaptable to WSNs, mainly because of the poor processing power of sensor nodes. For example Liu and Ning state in [14] that public-key cryptography is not feasible for sensor nodes; many similar statements can be found in other early papers on WSN security.

In 2004, Gura et al [8] published a now-famous paper on efficient implementation of public-key cryptography on the ATmega128, an 8-bit micro-controller used in many wireless sensor nodes, e.g. the MICAz mote [3]. By exploiting the large number of general-purpose registers of the AVR architecture, they developed a new technique for speeding up the multiplication of multiple-precision integers, the nowadays widely-used *hybrid method*. Hybrid multiplication reduces the number of memory accesses (`ld` instructions), which, in turn, considerably decreases the overall execution time of a full modular exponentiation or a full scalar multiplication

Proceedings of the 1st International Conference on Computer Science and Network Technology (ICCSNT 2011), vol. 1, pp. 459–466.

on an elliptic curve. Gura et al reported an execution time of 3106 clock cycles for a (160×160)-bit multiplication on the ATmega128. They also implemented a full elliptic curve scalar multiplication over a standardized 160, 192, and 224-bit prime field, for which they specified an execution time of 0.81, 1.24, and 2.19 seconds, respectively (measured at a clock frequency of 8 MHz). Gura et al's work had a massive impact on research in WSN security because their results convincingly prove that Elliptic Curve Cryptography (ECC) [9] is feasible on resource-constrained sensor nodes. In the recent past, a large amount of research has been devoted to further improve the timings reported by Gura et al; notable work includes that of Uhsadel et al [23] (who managed to reduce the running time of a (160×160)-bit multiplication to 2881 clock cycles), Liu et al [16] (2865 cycles), Scott and Szczechowiak [21] (2651 cycles with "unrolled" loops), and Hutter et al [10] (2395 cycles). However, several of these research activities focused solely on fast multiple-precision multiplication and ignored all other arithmetic operations needed in ECC, in particular modular reduction.

In this paper, we introduce a high-speed implementation of arithmetic operations (addition, subtraction, multiplication and squaring) in *Optimal Prime Fields (OPFs)* for the ATmega128 and similar AVR-based 8-bit processors. OPFs were first described in [5] as a family of "low-weight" prime fields that allow for efficient software implementation of all operations requiring a modular reduction, in particular the field-multiplication. More formally, an OPF is a finite field defined by a prime of the form $p = u \cdot 2^k + v$, whereby the two coefficients u and v are "small" (in relation to 2^k) so that they fit into one (or a few) general-purpose registers of the target processor. In our implementation for the ATmega128, u is an integer with a length of at most 16 bits (i.e. u can be held in two 8-bit registers), while v is always 1. A concrete example is $p = 52542 \cdot 2^{144} + 1$ (i.e. $u = 52542$ and $v = 1$), which happens to be a 160-bit prime. Primes of such form are characterized by low Hamming weight as the binary representation of p contains only a few non-zero bits. More precisely, when the prime p is stored in an array of 8-bit words, only the two most significant words and the least significant word are non-zero. All the "middle" words of the array are zero and do not need to be processed in the modular reduction operation, which allows one to optimize the field arithmetic. In order to demonstrate the advantages and capabilities of OPFs in practice, we describe dedicated algorithms for Montgomery multiplication and squaring on basis of Gura et al's hybrid technique. Montgomery reduction modulo a low-weight prime of the form $p = u \cdot 2^k + v$ is a linear operation and, therefore, considerably faster than the reduction modulo a general prime. Our implementation for the ATmega128 is able to perform a multiplication in a 160-bit OPF in merely 3,542 clock cycles, which represents a new speed record for 160-bit modular multiplication on an 8-bit processor.

2. OPTIMAL PRIME FIELDS (OPFS)

Elliptic curve cryptosystems utilize a group of points on an elliptic curve to derive variants of "classical" DLP-based public-key schemes such as DSA or Diffie-Hellman [9]. The curves used in ECC are defined over a finite field, which is typically either a prime field or a binary extension field. In the former case, implementers often choose special primes (e.g. generalized-Mersenne (GM) primes [18]) to reduce the cost of the modular reduction operation [12]. However, the implementation described in this paper is based on a novel family of prime fields, the so-called optimal prime fields.

Optimal Prime Fields (OPFs) were originally introduced in [5] as finite fields defined through a prime p of the form $u \cdot 2^k + v$, where u and v are small coefficients that fit into a single register of the target processor (or, in more formal terms, $0 < u, v < 2^w$ with w denoting the processor's word size). However, for reasons of efficiency, we fix v to 1 since this allows for special optimization of the Montgomery reduction, as will be explained in Section 4. The ATmega128 is an 8-bit processor, and, consequently, u has to be in the range of $0 < u < 256$ according to the original definition of OPFs [5]. Unfortunately, the number of OPFs having an order of 160 bits (which is a typical field-order for ECC in WSNs [13]) is relatively small when u is restricted to have a bitlength of (at most) 8 bits so that it fits into a single general-purpose register. In order to have a larger choice of 160-bit OPFs, we "soften" the original condition for the selection of u specified in [5] and allow u to be up to 16 bits long (i.e. u occupies two registers on a ATmega128 instead of one). An example for one of the special primes we use in the present paper is $p = 52542 \cdot 2^{144} + 1$, which happens to be a 160-bit prime that looks as follows when written in hex notation:

$$p = \text{0xCD3E0000000000000000000000000000000000001}$$

As mentioned in Section 1, OPFs are characterized by a low Hamming weight [5]. For example, when p is stored in an array of 8-bit words, only the two most significant words and least significant word are non-zero; all other words in between them are 0. The low weight of these primes allows for very efficient implementation of the modular reduction operation since only the three non-zero words of p need to be processed, as will be shown in detail in the following two sections. Well-known modular reduction methods, such as Montgomery [17] and Barrett reduction, can be optimized for low-weight primes so that the reduction operation has linear complexity, similar to the generalized-Mersenne (GM) and pseudo-Mersenne (PM) primes [9, 22]. A special advantage of OPFs over GM prime fields is their flexibility; there exist a large number of 160-bit OPFs (and also OPFs of a length of 192, 224, and 256 bits) if we allow u to be up to 16 bits long, whereas the number of GM primes with good arithmetic properties is rather limited. We refer to [7] for an in-depth discussion of the advantages of OPFs.

We implemented multiplication and squaring for OPFs of order 160, 192, 224, and 256 bits using Montgomery's modular reduction method in combination with Gura et al's hybrid technique. More precisely, we optimized the so-called Finely Integrated Operand Scanning (FIPS) method [11, 6] for Montgomery modular multiplication with respect to the low weight of our OPFs so that only the three non-zero bytes (i.e. the two MSB and the LSB) of the prime are processed during the reduction step. An implementation of the FIPS method for normal primes executes $2s^2 + s$ single-precision multiplications (i.e. `mul` instructions) when the operands are s bytes long[1]. In the case of processing 160-bit operands on an 8-bit processor (i.e. $s = 20$), this amounts to an overall

[1]Note that the number of single-precision multiplications is not affected by the hybrid technique. Hybrid multiplication reduces the total number of memory references, but not the number of `mul` instructions.

number of 820 `mul` instructions. However, when the FIPS technique is optimized taking into account that $s-3$ bytes of p are zero and the LSB is 1, only s^2+2s multiplications have to be carried out, which results in 440 `mul` instructions for 160-bit operands. A conventional multiplication of two s-byte operands (without reduction) requires s^2 `mul` instructions; consequently, the "overhead" of modular reduction in an OPF is $2s$ `mul` instructions, i.e. the reduction cost scales linearly with the operand length.

3. MULTIPLE-PRECISION ARITHMETIC

In this section, we describe a number of basic algorithms for multiplication, squaring, modular reduction, as well as modular multiplication for multiple-precision integers. We first introduce some terminology and notations that will be used throughout this paper. Multiple-precision integers are denoted by capital italic letters, e.g. A. An n-bit multiple-precision integer can be stored in an array of w-bit words (digits), where w denotes the number of bits per word. It is common practise to choose w such that it corresponds to the word length of the target processor, which means w is either 8 (as in our case), 16, or 32. The letter s represents the total number of words that an n-bit multiple-precision integer contains, i.e. $s = \lceil n/w \rceil$. We use indexed lowercase letters a_i to denote the individual words of A. Hence,

$$A = \sum_{i=0}^{s-1} a_i \cdot 2^{iw} \quad \text{with} \quad 0 \leq a_i \leq 2^w - 1. \tag{1}$$

The array of w-bit words that represents a multi-precision integer A is $\{a_{s-1}, a_{s-2}, \ldots, a_1, a_0\}$, whereby a_0 refers to the Least Significant Word (LSW) and accordingly a_{s-1} is the Most Significant Word (MSW). Normally, we use A, B as operands, N as modulus and P as final result.

3.1 Multiplication

We sketch three basic techniques for the implementation of multiple-precision multiplication: the schoolbook method [9], Comba's method [2], and the hybrid method [8].

Schoolbook Method

The most straightforward way to obtain the product of two multiple-precision integers is the *schoolbook method*, which is similar to the algorithm for multiplying multi-digit numbers taught in elementary school. The schoolbook method has a nested-loop structure with a relatively tight inner loop (see Algorithm 2.9 in [9]). In each iteration of the inner loop, an operation of the form $c_{i+j} + a_i \cdot b_j + u$ is carried out, i.e. a word a_i of operand A is multiplied by a word b_j of operand B, and two other words, namely c_{i+j} and u, are added to the $2w$-bit product $a_i \cdot b_j$. Each iteration of the outer loop multiplies a_i by the s words of B, starting with b_0 [9]. The schoolbook method is also called *operand-scanning* method since the outer loop moves through the words of one of the operands [11].

Comba's Method

Comba's method, originally introduced in [2], is also known as *product-scanning* method because the outer loop moves through the words of the product. It actually executes two nested loops (see Algorithm 2.10 in [9]); the first computes the lower s words of the final product, whereas the second yields the higher s words. Both nested loops perform the same inner-loop operation, namely a Multiply-ACcumulate (MAC) operation of the form $S + a_i \cdot b_j$, i.e. two words are multiplied and the $2w$-bit product is added to a cumulative sum S. This sum will normally exceed $2w$ bits when the inner loop is iterated two or more times, which means we need three w-bit registers to store S [9]. After termination of the inner loop, the least significant word of S is a word of the final product and can be written to memory.

The performance of Comba's method is often better than that of the schoolbook technique because it executes less memory store operations. In schoolbook multiplication, all words of the (intermediate) product, except the LSW and MSW, are loaded from and written back to memory several times. On the other hand, Comba's method writes a word of the final product to memory only once, namely after its complete evaluation (i.e. after termination of the inner loop) [6]. Comba multiplication executes store instructions in the outer loop, but not in the inner loop. However, a disadvantage of Comba's method is the bitlength of the cumulative sum S, which makes an implementation in C or Java quite difficult since these programming languages do not provide a triple-precision data type.

Hybrid Method

Hybrid multiplication [8] is not really a new multiplication technique, but rather an ingenious optimization of Comba's method for processors with a large number of registers. The principal structure of the hybrid method is the same as in Comba's method, which means that the algorithm consists of two nested loops. When performing an ordinary Comba multiplication, one word of A and one word of B are loaded from data memory and multiplied together in each iteration of the inner loop. On the other hand, the hybrid method loads $d \geq 2$ words of each operand and performs d^2 multiplications per inner-loop iteration, which reduces the total number of iterations from s^2 to $(\frac{s}{d})^2$. This, in turn, reduces the overall number of load operations by a factor of d since only $2d$ loads are carried out in each iteration. The concrete value of d (i.e. the number of words of A and B that are processed per iteration) is determined by the number of free registers on the target processor; on the AVR platform it is common practice to choose $d = 4$ [8].

The inner loop of the hybrid method multiplies d words of operand A by d words of operand B (i.e. it performs a $(d \times d)$-word multiplication) and adds the resulting $2d$-word product to a running sum held in $2d+1$ registers. When working on an AVR processor, a w-bit word is nothing else than a byte, and the inner-loop operation boils down to multiplying 4 bytes of A by 4 bytes of B. This (4×4)-byte multiplication and the addition of the 8-byte product to a 9-byte sum can be carried out in many different ways; the original hybrid method of Gura et al from [8] employs the schoolbook method for it (shown on the left of Figure 1). In 2010, Liu et al [16] came up with an alternative approach to perform the inner-loop operation, which is depicted in the middle of Figure 1. In their implementation, the 16 byte-products (i.e. the products $a_0 \cdot b_0$ to $a_3 \cdot b_3$) are calculated in a non-conventional order with the goal of reducing the number of `add`/`adc` and `mov`/`movw` instructions compared to the original inner-loop operation of Gura et al.

Our implementation of the inner-loop operation is based on that of Liu et al, but we perform the computation of the last four byte-products is a different order, which allows us

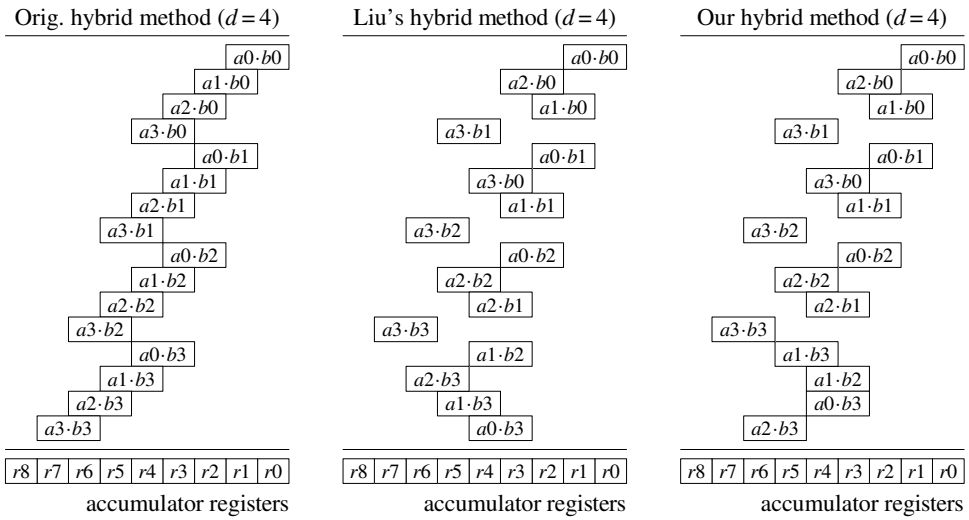

Figure 1: Inner-loop operation of Gura et al's original hybrid method (left), Liu et al's improved version (middle), and our implementation (right).

to save one `movw` instruction. As can be observed from the right of Figure 1, we process the first 12 byte-products in the same way as described in [16]. The last four byte-products (i.e. $a_1 \cdot b_3$, $a_1 \cdot b_2$, $a_2 \cdot b_3$, $a_0 \cdot b_3$) are generated and added to the accumulator registers (r_0 to r_8) as follows: we first multiply a_1 by b_3 and move the resulting product to two temporary registers via the `movw` instruction. Then, we calculate the product $a_1 \cdot b_2$ and add the lower byte to the content of register r_3. The upper byte is added (with carry) to the temporary register pair holding $a_1 \cdot b_3$. Note that this addition can not produce a "carry out," i.e. this addition can not overflow the temporary register pair. The subsequent product $a_0 \cdot b_3$ is processed in the same way, i.e. the lower byte is added to r_3 and the higher-order byte to the two temporary registers (again without overflow). After the final multiplication of a_2 by b_3, the lower temp register is added to r_4; a possibly resulting carry bit is added with the upper temp register to the product $a_2 \cdot b_3$. The obtained sum is then added to the accumulator registers r_5 to r_8. In summary, the processing of these four byte products takes four `mul`, one `movw`, and 13 `add` (or `adc`) instructions. The complete inner-loop operation for $d = 4$ amounts to a total of 46 `add` (resp. `adc`), 16 `mul`, eight `ld` (i.e. load), and seven `movw` instructions.

3.2 Squaring

Theoretically, squaring is almost twice as fast as a normal multiplication. When performing a schoolbook multiplication (see Algorithm 2.9 in [9]) with two operands that are the same (i.e. $A = B$), then any partial product of the form $a_i \cdot b_j$ is identical to $a_j \cdot b_i$. In fact, many partial products would be calculated twice when an ordinary multiplication algorithm (e.g. the schoolbook method) is used to square an integer. An optimized squaring algorithm takes this into account so that each partial product is calculated only once and then doubled (via a left-shift) if needed. However, the partial products of the form $a_i \cdot b_j$ with $i = j$ appear only once and do not need to be doubled.

Since Comba's multiplication technique is faster than the schoolbook method, we only consider Comba squaring. We compute all partial products $a_i \cdot a_j$ with $i \neq j$ only once but add them twice to the cumulative sum S, whereas the products of the form a_i^2 (i.e. $a_i \cdot a_j$ with $i = j$) are added normally since they do not appear twice in the computation of the square. Therefore, we need an `if-then` statement to distinguish between these two cases. This conditional statement costs extra cycles, but the larger the operands get the more instruction are saved by this optimization.

3.3 Montgomery Reduction

Modular multiplication, i.e. an operation of the form $P = A \cdot B \bmod N$, is relatively slow since modular reduction is a costly operation that would normally require a division. In 1985, Peter Montgomery introduced an efficient algorithm for modular reduction that replaces the trial division by a subtraction of a multiple of N and a right-shift operation [17]. The so-called *Montgomery multiplication* consists of a multiplication of two integers and a Montgomery reduction of the product. In essence, Montgomery's algorithm allows one to efficiently compute the *Montgomery product* of two integers, which is defined as follows:

$$\text{MonPro}(A, B) \;=\; A \cdot B \cdot R^{-1} \bmod N \qquad (2)$$

The factor R, called Montgomery radix, is typically chosen to be a power of two, e.g. $R = 2^n$ where n denotes the size of N in bits, so that the multiplication by R^{-1} is simply a shift operation (see [17] for an in-depth description).

Koç et al present in [11] five algorithms for computation of the Montgomery product in software. These algorithms are classified by two criteria; the first one is whether multiplication and reduction are integrated or separated, and the second criterium is whether the multiplication is based on the schoolbook method (i.e. operand scanning) or Comba's method (i.e. product scanning). An example for the latter is the *Finely Integrated Product Scanning (FIPS)* method (as shown in Algorithm 1), which is basically Comba's method with "finely" integrated Montgomery reduction. The FIPS method has two nested loops (similar to Comba's method) and performs two MAC operations in the inner loop. More precisely, in each iteration of the inner loop, two products

(namely $a_j \cdot b_{i-j}$ and $p_j \cdot n_{i-j}$ in Algorithm 1) are added to a cumulative sum S. This sum is held in the three registers t, u and v, i.e. (t, u, v) denotes a $3w$-bit word. The words of A and P are loaded in ascending order (i.e. from less to more significant positions), while the words of B and N are loaded in descending order.

Algorithm 1: FIPS Montgomery multiplication

Input: n-bit modulus N, $2^{n-1} \leq N < 2^n$, two operands $A, B < N$, pre-computed constant $n'_0 = -n_0^{-1} \bmod 2^w$
Output: Montgomery product $P = A \cdot B \cdot 2^{-n} \bmod N$

1: $(t, u, v) \leftarrow 0$
2: **for** i from 0 by 1 to $s-1$ **do**
3: **for** j from 0 by 1 to $i-1$ **do**
4: $(t, u, v) \leftarrow (t, u, v) + a_j \cdot b_{i-j}$
5: $(t, u, v) \leftarrow (t, u, v) + p_j \cdot n_{i-j}$
6: **end for**
7: $(t, u, v) \leftarrow (t, u, v) + a_i \cdot b_0$
8: $p_i \leftarrow v \cdot n'_0 \bmod 2^w$
9: $(t, u, v) \leftarrow (t, u, v) + p_i \cdot n_0$
10: $v \leftarrow u$, $u \leftarrow t$, $t \leftarrow 0$
11: **end for**
12: **for** i from s by 1 to $2s-2$ **do**
13: **for** j from $i-s+1$ by 1 to $s-1$ **do**
14: $(t, u, v) \leftarrow (t, u, v) + a_j \cdot b_{i-j}$
15: $(t, u, v) \leftarrow (t, u, v) + p_j \cdot n_{i-j}$
16: **end for**
17: $p_{i-s} \leftarrow v$
18: $v \leftarrow u$, $u \leftarrow t$, $t \leftarrow 0$
19: **end for**
20: $p_{s-1} \leftarrow v$
21: $p_s \leftarrow u$
22: **if** $P \geq N$ **then** $P \leftarrow P - N$ **end if**

4. IMPLEMENTATION FOR ATMEGA128

Since we work on an ATmega128 processor, the word size w is 8 bits (i.e. one byte) in our case. The ATmega128 has a memory space of 64 kB and, hence, two bytes are needed to represent an address. All implementations we describe in the following process $d = 4$ bytes "at once," i.e. four bytes of operand A and operand B are loaded per iteration of the inner loop. We divide each byte-array representing a multi-precision integer into groups of four bytes, starting at the least significant byte. These groups of bytes are referred to using indexed uppercase letters, e.g. the i-th group of bytes of operand A is A_{i-1}. An s-byte operand can be split into $k = \lceil s/4 \rceil$ groups; the first (i.e. least significant) group is A_0 and consists of the four bytes a_3, a_2, a_1, and a_0.

4.1 Modular Addition and Subtraction

In order to calculate the modular sum $A + B \bmod N$, we firstly perform the addition and then do the reduction. It is not necessary to always completely reduce the result since we work in a prime field. Our implementation subtracts the modulus N from the sum until we obtain a result that has the same bitlength as N, even if it is not fully reduced. Note that we also accept incompletely-reduced operands, i.e. we do not insist that $A, B < N$, but the bitlength of these two operands must not exceed n, the bitlength of N.

The addition starts with the two least significant groups (i.e. A_0 and B_0). In the next step, the sum $A_1 + B_1 + c$ is calculated, whereby c denotes the carry bit generated in the first addition of 4-byte groups. After adding up the last two groups (i.e. $A_{k-1} + B_{k-1} + c$), the addition is complete and we have a sum that is up to $n + 1$ bits long. A reduction (i.e. subtraction of N) is necessary when the bitlength of the sum exceeds that of N, which is actually the case when the addition has produced a "carry out." Note that up to two subtractions of N may be needed to get an n-bit result since the operands A and B are not necessarily fully reduced.

The modular subtraction $A - B \bmod N$ is very similar to the modular addition, except that for the reduction up to two additions of N have to be carried out.

4.2 Hybrid Multiplication and Squaring

As explained in Section 3.1, the hybrid method executes two nested loops; the structure of the loops is similar as in Algorithm 1, except that only one MAC operation is carried out in each loop iteration. We perform this MAC operation according to Section 3.1, i.e. one 4-byte group of A and one 4-byte group of B are multiplied together to yield an 8-byte product, which is added to a running sum consisting of nine bytes. However, there are two "special" cases for which we implemented the MAC operation in a different fashion. One is the very first iteration of the inner loop, in which A_0 is multiplied by B_0. We notice that in the beginning, the value in the accumulator (consisting of 9 registers) is zero. Therefore, the MAC operation can be replaced by a conventional multiplication, i.e. we move the byte products directly into the accumulator instead of adding them to it, which saves several `add` or `adc` instructions. Our special implementation of $A_0 \cdot B_0$ needs 36 `add/adc` and ten `movw` instructions on an ATmega128. The other special case is the first iteration of the two inner loops, i.e. the computation $A_0 \cdot B_i$ in the first inner loop and $A_j \cdot B_{k-1}$ in the second inner loop. Note that when j is 0 or $i - s + 1$, the four most significant accu registers are zero due to a previously executed shift of the accumulator. When adding byte products to the accu, the propagation of carries can stop at the first register whose value is 0 instead of register $r8$. Our implementation of this special MAC operation executes a total of 40 `add/adc` and ten `movw` instructions.

Hybrid squaring also uses two nested loops to compute a square $P = A \cdot A$. As mentioned in Section 3.2, we need an `if` statement to find out whether an 8-byte product of the form $A_i \cdot A_j$ must be added once or twice to the cumulative sum. In fact, the products of the form $A_i \cdot A_i$ (which are added only once) appear only in those 4-byte groups of the final result P that have an even index i, e.g. P_0, P_2, and so on. Therefore, the condition that the `if` statement has to check is simply whether the least significant bit of i is zero or not. As in hybrid multiplication, the very first iteration of the inner loop (in which $A_0 \cdot A_0$ is computed) allows for a special optimization of the MAC operation since the accu registers are 0. The situation is similar for the last iteration in which $A_{k-1} \cdot A_{k-1}$ is added to a cumulative sum whose four MSBs are zero. We "peeled off" the very first and the very last iteration from the corresponding loops so that we can utilize optimized MAC operations as described above for hybrid multiplication. However, it is not easily possible to optimize the first iteration of the inner loops of hybrid squaring. For example, if we compute $A_0 \cdot A_i$ "outside" the inner loop, then actually two iterations are taken out from the loop, which means we need extra instructions to decide

whether there are still multiplications to be performed in the inner loop. These extra instructions take more cycles than what could be saved by an optimized MAC operation for the first iteration.

4.3 Modular Multiplication in OPFs

The standard FIPS method, shown in Algorithm 1, adds in each iteration of the inner loop $a_j \cdot b_{i-j}$ and $p_j \cdot n_{i-j}$ to a cumulative sum (t, u, v). Unfortunately, AVR processors have only three pointer registers, which is does not suffice for the four operands A, B, N, and P. Therefore, we have to use one pointer register to point to two multi-precision integers and use `push` and `pop` to keep one address on the stack while executing the inner loop. This requires many instructions that do not contribute to the final result.

Algorithm 2: OPF-FIPS Montgomery multiplication

Input: n-bit modulus $N = (n_{s-1}, 0, 0, \ldots, 0, 1)$ represented by an array of s words, two operands $A, B < N$
Output: Montgomery product $P = A \cdot B \cdot 2^{-n} \bmod N$

1: $(t, u, v) \leftarrow 0$
2: **for** i from 0 by 1 to $s-1$ **do**
3: **for** j from 0 by 1 to i **do**
4: $(t, u, v) \leftarrow (t, u, v) + a_j \cdot b_{i-j}$
5: **end for**
6: **if** $i = s-1$ **then** $(t, u, v) \leftarrow (t, u, v) + p_0 \cdot n_{s-1}$ **end if**
7: $p_i \leftarrow -v \bmod 2^w$
8: $(t, u, v) \leftarrow (t, u, v) + p_i$
9: $v \leftarrow u$, $u \leftarrow t$, $t \leftarrow 0$
10: **end for**
11: **for** i from s by 1 to $2s-2$ **do**
12: **for** j from $i-s+1$ by 1 to $s-1$ **do**
13: $(t, u, v) \leftarrow (t, u, v) + a_j \cdot b_{i-j}$
14: **end for**
15: $(t, u, v) \leftarrow (t, u, v) + p_{i-s+1} \cdot n_{s-1}$
16: $p_{i-s} \leftarrow v$
17: $v \leftarrow u$, $u \leftarrow t$, $t \leftarrow 0$
18: **end for**
19: $p_{s-1} \leftarrow v$
20: $p_s \leftarrow u$
21: **if** $P \geq N$ **then** $P \leftarrow P - N$ **end if**

Our modulus N is an "optimal prime" as introduced in Section 2, which means that most of its bytes are just zero except the two most significant bytes and the least significant bit. If n_{i-j} is zero, then the product $p_j \cdot n_{i-j}$ is also zero and $(t, u, v) \leftarrow (t, u, v) + p_j \cdot n_{i-j}$ does not need to be executed in the inner loop. Eliminating this operation from the loop saves roughly 100 cycles. As our implementation processes four bytes at a time, only N_0 and N_{k-1} contain non-zero bytes. N_0 can be thought as n_0 in standard FIPS (see Algorithm 1), and N_{k-1} correspond to n_{s-1}. We have to first determine in which loop iterations n_0 and n_{s-1} are used as operands for a multiplication, and then modify the algorithm to "peel off" these iterations from the loop. Since n_0 is 1 when using our low-weight primes, we do not need to multiply p_i by n_0; instead, we can directly add p_i to the cumulative sum in (t, u, v). On the other hand, n_{s-1} is used in both of the nested loops. In the first nested loop, n_{s-1} is loaded if and only if $i = s-1$ and $j = 0$, i.e. in the final iteration of the outer loop. In this last iteration, $p_0 \cdot n_{s-1}$ is computed once. In the second nested loop, $i - j$ (which is the index of N) becomes $s-1$ in the first iteration of the inner loop and the inner loop is executed in each iteration of the outer loop. Therefore, the operation $p_{i-s+1} \cdot n_{s-1}$ is carried out $s-1$ times in the second nested loop.

Our optimized variant of the FIPS technique for OPFs is shown in Algorithm 2. The only operation performed in the inner loops is $(t, u, v) \leftarrow (t, u, v) + a_j \cdot b_{i-j}$ and, hence, no stack operations (i.e. `push`, `pop`) are needed, which makes our FIPS variant very efficient. Since the modulus N is one of our special primes, we only pass the most significant two bytes of N as parameter to the Assembly function that implements the FIPS multiplication; these two bytes can be kept in two registers of the ATmega128. Therefore, we can use the three available pointer registers to hold the address of A, B, and P. Even though Algorithm 2 shows a standard (i.e. non-hybrid) version of the FIPS method for OPFs, we actually process four bytes at once, very similar to Section 4.2. As $(t, u, v) \leftarrow (t, u, v) + p_0 \cdot n_{s-1}$ is only executed once in the first nested loop, we apply "loop peeling" to perform this operation between the two nested loops (line 6 of Algorithm 2); this saves $k-1$ executions of the `if` statement in line 6. Furthermore, we can ignore the least significant two bytes of N_{k-1} when multiplying P_{i-k+1} by N_{k-1} because these bytes are 0. A further optimization is possible due to the fact that N_0 (i.e. the least significant 4-byte group of N) is 1, which implies N'_0 is $2^{32} - 1$. As a consequence, we can replace the operation $v \cdot n'_0 \bmod 2^w$ in line 8 of Algorithm 1 by a computation of the two's complement of v, which, in our case, requires to compute the two's complement of the least significant 4-byte group of the accumulator.

We also optimized the final subtraction (line 21 in Algorithm 2), taking into consideration that our modulus N is a special prime of the form $u \cdot 2^k + 1$. We first subtract 1 from P_0 if the product P is longer than N (this possible excess bit, which we call carry bit, is stored in byte p_s of P). If the subtraction $P_0 - 1$ does not generate a "borrow bit," then we directly jump to the most significant four bytes of P and N and subtract N_{k-1} from P_{k-1}. On the other hand, if a borrow bit was generated (which can only happen when the least significant four bytes of P are all zero), we perform a normal subtraction with borrow. This subtraction begins with $P_1 - N_1 - 1$ and ends with $P_{k-1} - N_{k-1} - b$, whereby b (the borrow bit) either 1 or 0.

4.4 Modular Squaring in OPFs

All partial products of the form $a_i \cdot a_j$ that appear twice in the squaring operation are computed only once and then added twice to the accumulator. In the conventional FIPS multiplication, the two products $a_j \cdot b_{i-j}$ and $p_j \cdot n_{i-j}$ are processed together in the same inner loop. However, when $A = B$, the products of the form $p_i \cdot n_j$ and $p_j \cdot n_i$ are still different, which makes FIPS squaring quite complicated to implement.

Given the "special" form of our modulus N, most of the products $p_i \cdot n_j$ are 0 and can be ignored. The inner loops contribute to the computation of the square A^2, but not to the reduction since the few non-zero products of the form $p_j \cdot n_{i-j}$ are processed outside the inner loop. The situations where n_{s-1} and n_0 are used as operands are the same as in OPF-FIPS multiplication (Section 4.3). We only perform $(t, u, v) \leftarrow (t, u, v) + p_0 \cdot n_{s-1}$ once in the last iteration of the first outer loop, while $(t, u, v) \leftarrow (t, u, v) + p_{i-s+1} \cdot n_{s-1}$ is executed in each iteration of the second outer loop. Nonetheless, we still need an `if` statement to decide whether the

Instr. type	add	mul	ld	st	mov	Other	Total
CPI	1	2	2	2	1	cycles	cycles
160 bits	1092	400	200	40	202	271	2845
192 bits	1586	576	288	48	285	354	4049
224 bits	2172	784	392	56	382	443	5461
256 bits	2850	1024	512	64	493	538	7081

Table 1: Instruction counts of hybrid multiplication

Instr. type	add	mul	ld	st	mov	Other	Total
CPI	1	2	2	2	1	cycles	cycles
160 bits	974	240	120	40	121	376	2271
192 bits	1400	336	168	48	168	494	3166
224 bits	1902	448	224	56	223	620	4201
256 bits	2480	576	288	64	286	754	5376

Table 2: Instruction counts of hybrid squaring

Instr. type	add	mul	ld	st	mov	Other	Total
CPI	1	2	2	2	1	cycles	cycles
160 bits	1272	440	220	40	232	460	3364
192 bits	1802	624	312	48	321	579	4670
224 bits	2424	840	420	56	424	704	6184
256 bits	3138	1088	544	64	493	883	7906

Table 3: Instruction counts of hybrid Montgomery multiplication in OPFs (without final subtraction)

Instr. type	add	mul	ld	st	mov	Other	Total
CPI	1	2	2	2	1	cycles	cycles
160 bits	1154	280	140	40	151	589	2814
192 bits	1616	384	192	48	204	754	3822
224 bits	2154	504	252	56	265	928	4971
256 bits	2768	640	320	64	334	1110	6260

Table 4: Instruction counts of hybrid Montgomery squaring in OPFs (without final subtraction)

Op. length	Library	Min.	Max.	Avg.
160 bits	TinyECC ($d = 5$)	3243	3890	3568
	WM-ECC	3356	3631	3534
	Our work	3006	3006	3006
192 bits	TinyECC ($d = 4$)	4649	5561	5051
	Our work	4210	4210	4210
224 bits	TinyECC ($d = 4$)	6229	7481	6766
	Our work	5622	5622	5622
256 bits	TinyECC ($d = 4$)	8043	9690	8715
	Our work	7242	7242	7242

Table 5: Execution time (in clock cycles) of multiplication for different operand lengths

Op. length	Library	Min.	Max.	Avg.
160 bits	TinyECC ($d = 4$)	3010	3175	3092
	WM-ECC	3228	3234	3234
	Our work	2428	2428	2428
192 bits	TinyECC ($d = 4$)	4198	4436	4297
	Our work	3323	3323	3323
224 bits	TinyECC ($d = 4$)	5586	5905	5702
	Our work	4358	4358	4358
256 bits	TinyECC ($d = 4$)	7174	7582	7307
	Our work	5533	5533	5533

Table 6: Execution time (in clock cycles) of squaring for different operand lengths

Op. length	Library	Min.	Max.	Avg.
160 bits	TinyECC ($d = 4$)	14625	15113	14929
	WM-ECC	3797	4071	3985
	Our work	3521	3588	3542
192 bits	TinyECC ($d = 4$)	19408	20758	20060
	Our work	4827	4894	4851
224 bits	TinyECC ($d = 4$)	24872	26145	25765
	Our work	6520	6588	6545
256 bits	TinyECC ($d = 4$)	31054	32860	32258
	Our work	8063	8130	8091

Table 7: Execution time (in clock cycles) of modular multiplication for different operand lengths

Op. length	Library	Min.	Max.	Avg.
160 bits	TinyECC ($d = 4$)	14646	15123	14929
	WM-ECC	3669	3675	3675
	Our work	2966	3032	2990
192 bits	TinyECC ($d = 4$)	19408	20747	20060
	Our work	3974	4040	3999
224 bits	TinyECC ($d = 4$)	24872	26134	25765
	Our work	5123	5189	5148
256 bits	TinyECC ($d = 4$)	31054	32849	32258
	Our work	6412	6478	6438

Table 8: Execution time (in clock cycles) of modular squaring for different operand lengths

product $a_i \cdot a_i$ (which is not doubled) has to be computed or not. The condition for the `if` clause in the first nested loop is whether index i is even or not (similar as in Comba squaring), whereas the condition for the other `if` clause in the second nested loop is whether $s + i$ is even or not (this depends on both the loop index i and the length s of the operands). Our implementation of OPF-squaring processes four bytes at once (analogously to hybrid squaring) and we also "peeled off" the very first iteration (in which $A_0 \cdot A_0$ is computed) and the very last iteration (for $A_{k-1} \cdot A_{k-1}$) to allow for optimization of the MAC operations, taking into account that all (or some) bytes of the accumulator are zero (see Section 4.2 for further details). The final subtraction of N is carried out in the same way as in Section 4.3.

4.5 Experimental Results and Comparison

We measured the execution time of our implementation of hybrid multiplication and squaring, as well as OPF-FIPS multiplication and squaring, on an ATmega128 processor. In order to facilitate comparison with prior work, we provide timings for four operand lengths, namely 160, 192, 224, and 256 bits. Table 1 to 4 summarize the number of `add` (plus `adc`), `mul`, `ld`, `st`, and `movw` instructions executed by these four operations and also specify the overall execution time in clock cycles (excluding function-call overhead).

There exist a number of libraries for fast multi-precision arithmetic and ECC on the AVR platform; two well-known examples are TinyECC [13] and WM-ECC [24]. We detail the timings for multiplication, squaring, modular multiplication, and modular squaring of these libraries and our own library in Table 5 to 8. TinyECC uses Barrett's algorithm

for modular reduction and supports arbitrary primes, while our implementation only supports Montgomery reduction in OPFs. The operand size of WM-ECC is fixed to 160 bits [24], but TinyECC can handle all four operand sizes. Note that the multiplication time of TinyECC and WM-ECC is not constant for a given operand length, but depends on the form of the operands. For example, TinyECC features some optimizations that allow it to achieve lower execution time when the Hamming weight of the two operands is low than when it is high. Table 5 to 8 summarize both the minimal execution time (for low-weight operands) and the maximal execution time (when all bits of the operands are 1). The tables also list the average time needed for 100 executions of a given operation when using pseudo-random numbers as operands. Our implementation of of hybrid multiplication and squaring has a constant execution time. However, the execution time of OPF-FIPS multiplication and squaring is not constant but depends on whether a final subtraction is carried out or not. The difference in execution time between these two cases is small (max. 2%). Note that all timings in Table 5 to 8 include the full function-call overhead.

5. CONCLUSIONS

We presented a highly-optimized library for arithmetic in Optimal Prime Fields (OPFs) on 8-bit AVR processors like the ATmega128. Our library is fully parameterized in terms of operand length (i.e. it can process operands of any size) and contains all arithmetic operations needed in ECC. The modular reduction follows Montgomery's algorithm and is optimized for low-weight primes of the form $p = 2^k + 1$. We also developed an improved variant of the MAC operation carried out in the inner loop of the hybrid method and integrated it into the multiplication and squaring function for OPFs. Our implementation performs a multiplication in a 160-bit OPF in just 3,542 clock cycles on an ATmega128 processor, which sets a new speed record for 160-bit modular multiplication on 8-bit platforms. Compared to previous work, our implementations is roughly five times faster than the widely-used TinyECC (which needs 14,929 clock cycles for a 160-bit modular multiplication) and 11.1% faster than WM-ECC (which requires some 4,000 cycles). As our future work, we intend to protect the OPF library against Simple Power Analysis (SPA) attacks.

6. REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Çayirci. Wireless sensor networks: A survey. *Computer Networks*, 38(4):393–422, Mar. 2002.

[2] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Dec. 1990.

[3] Crossbow Technology, Inc. MICAz Wireless Measurement System. Data sheet, available for download at `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf`, Jan. 2006.

[4] S. R. Dussé and B. S. Kaliski. A cryptographic library for the Motorola DSP56000. In *Advances in Cryptology — EUROCRYPT '90*, vol. 473 of *Lecture Notes in Computer Science*, pp. 230–244. Springer Verlag, 1991.

[5] J. Großschädl. TinySA: A security architecture for wireless sensor networks. In *Proceedings of the 2nd International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2006)*, pp. 288–289. ACM, 2006.

[6] J. Großschädl and G.-A. Kamendje. Architectural enhancements for Montgomery multiplication on embedded RISC processors. In *Applied Cryptography and Network Security — ACNS 2003*, vol. 2846 of *Lecture Notes in Computer Science*, pp. 418–434. Springer Verlag, 2003.

[7] J. Großschädl et al. Optimal prime fields for use in elliptic curve cryptography. Unpublished manuscript.

[8] 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*, vol. 3156 of *Lecture Notes in Computer Science*, pp. 119–132. Springer Verlag, 2004.

[9] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

[10] M. Hutter and E. Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *Cryptographic Hardware and Embedded Systems — CHES 2011*, vol. 6917 of *Lecture Notes in Computer Science*, pp. 459–474. Springer Verlag, 2011.

[11] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[12] 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*, vol. 5746 of *Lecture Notes in Computer Science*, pp. 112–127. Springer Verlag, 2009.

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

[14] D. Liu and P. Ning. Establishing pairwise keys in distributed sensor networks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pp. 52–61. ACM Press, 2003.

[15] D. Liu and P. Ning. *Security for Wireless Sensor Networks*, vol. 28 of *Advances in Information Security*. Springer Verlag, 2006.

[16] Z. Liu, J. Großschädl, and I. Kizhvatov. Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers. In *Proceedings of the 1st International Workshop on the Security of the Internet of Things (SecIOT 2010)*, Tokyo, Japan, Nov. 2010.

[17] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.

[18] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). FIPS Publication 186-2, available online at `http://www.itl.nist.gov/fipspubs/`.

[19] Y. Qiu, J. Zhou, J. Baek, and J. Lopez. Authentication and key establishment in dynamic wireless sensor networks. *Sensors*, 10(4):3718–3731, Apr. 2010.

[20] C. S. Raghavendra, K. M. Sivalingam, and T. F. Znati. *Wireless Sensor Networks*. Kluwer Academic Publishers, 2004.

[21] 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`.

[22] J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR-99-39, Centre for Applied Cryptographic Research (CACR), University of Waterloo, Canada, 1999.

[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*, vol. 4572 of *Lecture Notes in Computer Science*, pp. 73–86. Springer Verlag, 2007.

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