Low-Weight Primes for Lightweight Elliptic Curve Cryptography on 8-bit AVR Processors

Zhe Liu[1], Johann Großschädl[1], and Duncan S. Wong[2]

[1] Laboratory of Algorithmics, Cryptology and Security,
University of Luxembourg, Luxembourg
{zhe.liu,johann.groszschaedl}@uni.lu
[2] Department of Computer Science,
City University of Hong Kong, Hong Kong SAR, China
duncan@cityu.edu.hk

Abstract. Small 8-bit RISC processors and micro-controllers based on the AVR instruction set architecture are widely used in the embedded domain with applications ranging from smartcards over control systems to wireless sensor nodes. Many of these applications require asymmetric encryption or authentication, which has spurred a body of research into implementation aspects of Elliptic Curve Cryptography (ECC) on the AVR platform. In this paper, we study the suitability of a special class of finite fields, the so-called *Optimal Prime Fields (OPFs)*, for a "lightweight" implementation of ECC with a view towards high performance and security. An OPF is a finite field $\mathbb{F}_p$ defined by a prime of the form $p = u \cdot 2^k + v$, whereby both u and v are "small" (in relation to 2^k) so that they fit into one or two registers of an AVR processor. OPFs have a low Hamming weight, which allows for a very efficient implementation of the modular reduction since only the non-zero words of p need to be processed. We describe a special variant of Montgomery multiplication for OPFs that does not execute any input-dependent conditional statements (e.g. branch instructions) and is, hence, resistant against certain side-channel attacks. When executed on an Atmel ATmega processor, a multiplication in a 160-bit OPF takes just 3237 cycles, which compares favorably with other implementations of 160-bit modular multiplication on an 8-bit processor. We also describe a performance-optimized and a security-optimized implementation of elliptic curve scalar multiplication over OPFs. The former uses a GLV curve and executes in 4.19 M cycles (over a 160-bit OPF), while the latter is based on a Montgomery curve and has an execution time of approximately 5.93 M cycles. Both results improve the state-of-the-art in lightweight ECC on 8-bit processors.

1 Introduction

The 8-bit AVR architecture [2] has grown increasingly popular in recent years thanks to its rich instruction set that allows for efficient code generation from high-level programming languages. A typical AVR microcontroller, such as the Atmel ATmega128 [3], features 32 general-purpose registers, separate memories

and buses for program and data, and some 130 instructions, most of which are executed in a single clock cycle. The AVR platform occupies a significant share of the worldwide smartcard market and other security-critical segments of the embedded systems industry, e.g. wireless sensor nodes. This has made AVR an attractive evaluation platform for research projects in the area of efficient implementation of cryptographic primitives for embedded devices. The literature contains papers dealing with block ciphers [10], hash functions [17], as well as public-key schemes based on Elliptic Curve Cryptography (ECC) [23]. Despite some recent progress [1, 5, 18], the implementation of ECC on 8-bit smartcards and sensor nodes is still a big challenge due to the resource constraints of these devices. A typical low-cost smartcard contains an 8-bit microcontroller clocked at a frequency of 5 MHz, 256 B RAM, and 16 kB ROM. On the other hand, a typical wireless sensor node, such as the MICAz mote [8], is equipped with an ATmega128 processor clocked at 7.3728 MHz and provides 4 kB RAM and 128 kB programmable flash memory.

1.1 Past Work on Lightweight ECC for 8-bit Processors

One of the first ECC software implementations for an 8-bit processor was presented by Woodbury et al in 2000 [41]. Their work utilizes a so-called Optimal Extension Field (OEF), which is a finite field consisting of p^m elements where p is a *pseudo-Mersenne prime* [7] (i.e. a prime of the form $p = 2^k - c$) and m is chosen such that an irreducible binomial $x(t) = t^m - \omega$ exists over GF(p). The specific OEF used in [41] is GF($(2^8 - 17)^{17}$) as this field allows the arithmetic operations, especially multiplication and inversion, to be executed efficiently on an 8-bit platform. Woodbury et al implemented the point arithmetic in affine coordinates and achieved an execution time of $23.4 \cdot 10^6$ clock cycles for a full 134-bit scalar multiplication on an 8051-compatible microcontroller that is significantly slower than the ATmega128. The first really efficient ECC software for an 8-bitter was introduced by Gura et al at CHES 2004 [15]. They reported an execution time of only $6.48 \cdot 10^6$ clock cycles for a full scalar multiplication over a 160-bit SECG-compliant prime field on the ATmega128. This impressive performance is mainly the result of a smart optimization of the multi-precision multiplication, the nowadays widely used *hybrid method*. In short, the core idea of hybrid multiplication is to exploit the large register file of the ATmega128 to process several bytes (e.g. four bytes) of the operands in each iteration of the inner loop(s), which significantly reduces the number of load/store instructions compared to a conventional byte-wise multiplication.

In the ten years since the publication of Gura et al's seminal paper, a large body of research has been devoted to further reduce the execution time of ECC on the ATmega128. The majority of research focussed on advancing the hybrid multiplication technique or devising more efficient variants of it. An example is the work of Uhsadel et al [37], who improved the handling of carry bits in the hybrid method and managed to achieve an execution time of 2881 cycles for a (160×160)-bit multiplication (without modular reduction), which is about 7.3% faster than Gura et al's original implementation (3106 cycles). Zhang et al [43]

re-arranged the sequence in which the byte-by-byte multiplications are carried out and measured an execution time of 2846 clock cycles. A further reduction of the cycle count to 2651 was reported by Scott et al [30], who fully unrolled the loops and used so-called "carry catcher" registers to limit the propagation of carries. This unrolled hybrid multiplication was adopted by Szczechowiak et al [35] to implement scalar multiplication over a 160-bit generalized-Mersenne prime field $\mathbb{F}_p$. An interesting result of their work is that the reduction modulo $p = 2^{160} - 2^{112} + 2^{64} + 1$ requires 1228 cycles, which means a full modular multiplication (including reduction) executes in 3882 clock cycles altogether. Also Lederer et al [22] came up with an optimized variant of the hybrid method and performed ECDH key exchange using the 192-bit generalized-Mersenne prime $p = 2^{192} - 2^{64} - 1$ as recommended by the NIST. A scalar multiplication needs $12.33 \cdot 10^6$ cycles for an arbitrary base point, and $5.2 \cdot 10^6$ cycles when the base point is fixed. The currently fastest means of multiplying two large integers on the ATmega128 is the so-called *operand-caching method* [19, 31], which follows a similar idea as the hybrid multiplication method, namely to exploit the large number of general-purpose registers to store (parts of) the operands.

Most lightweight ECC implementations for 8-bit AVR processors mentioned above suffer from two notable shortcomings, namely (1) they are vulnerable to side-channel attacks, e.g. *Simple Power Analysis (SPA)* [28], and (2) they make aggressive use of loop unrolling to reduce the execution time of the prime-field arithmetic, which comes at the expense of a massive increase in code size and poor scalability since the operand length is "fixed." SPA attacks exploit conditional statements and other irregularities in the execution of a cryptographic algorithm (e.g. double-and-add method for scalar multiplication [6]), which can leak key-related information through the power-consumption profile of a device executing the algorithm. However, not only the scalar multiplication, but also the underlying field arithmetic can be vulnerable to SPA attacks, e.g. due to conditional subtractions in the modular addition [34], modular multiplication [38], or modular reduction for generalized-Mersenne primes [29]. It was shown in various papers that SPA attacks on unprotected (or insufficiently protected) implementations of ECC pose a real-world threat to the security of embedded devices such as smart cards [25] or wireless sensor nodes [9].

Loop unrolling is a frequently employed optimization technique to increase the performance of the field arithmetic operations, in particular multiplication [1]. The basic idea is to replicate the loop body n times (and adjust the overall number of iterations accordingly) so that the condition for loop termination as well as the branch back to the top of the loop need to be performed only once per n executions. Full loop unrolling may allow some extra optimizations since the first and the last iteration of a loop often differ from the "middle" ones and can, therefore, be specifically tuned. However, full loop unrolling, when applied to operations of quadratic complexity (e.g. multiplication), bloats the code size (i.e. the size of the binary executable) significantly. Moreover, a fully unrolled implementation can only process operands up to a length corresponding to the number of loop iterations, which means it is not scalable anymore.

1.2 Contributions of this Paper

We present an efficient prime-field arithmetic library for the 8-bit AVR architecture that we developed under consideration of the resource constraints and security requirements of smart cards, wireless sensor nodes, and similar kinds of embedded devices. Our goal was to overcome the drawbacks of most existing implementations mentioned in the previous subsection, and therefore we aimed for a good compromise between performance, code size, and resistance against SPA attacks. Instead of using a Mersenne-like prime field, our library supports so-called *Optimal Prime Fields (OPFs)* [12] since this family of fields has some attractive properties that allow for efficient arithmetic on a wide range of platforms. An OPF is a finite field defined by a "low-weight" prime p of the form $p = u \cdot 2^k + v$, where u and v are small (in relation to 2^k) to that they fit into one or two registers of an 8-bit processor. The reduction modulo such a prime can be performed efficiently using Montgomery's algorithm [26] since only the non-zero bytes of p need to be processed. Our implementation is based on the OPF library from [43], but we significantly improved the execution time of all arithmetic operations (especially multiplication) and made it resistant against SPA attacks. We present a new variant of Montgomery modular multiplication for OPFs that does not perform any data-dependent indexing or branching in the final subtraction. A multiplication (including modular reduction) in a 160-bit OPF takes 3237 clock cycles on the ATmega128, which compares very well with previous work on modular multiplication for 8-bit processors.

Our OPF library uses an optimized variant of Gura et al's hybrid technique [15] for the multiplication, whereby we process four bytes of the two operands per iteration of the inner loop(s). However, in contrast to the bulk of previous implementations, we do not fully unroll the loops in order to keep the code size small. All arithmetic functions provided by our OPF library are implemented in a parameterized fashion and with "rolled" loops, which means that the length of the operands is not fixed or hard-coded, but is passed as parameter to the function along with other parameters such as the start address of the arrays in which the operands are stored. Consequently, our OPF library can support operands of varying length, ranging from 64 to 2048 bits (in 32-bit steps). This feature makes our OPF library highly scalable since one and the same function can be used for operands of different length without re-compilation.

We provide benchmarking results for operand lengths of 160, 192, 224, and 256 bits on the 8-bit ATmega128 processor, which we obtained with help of the cycle-accurate simulator of AVR Studio. For the purpose of benchmarking, we also implemented and simulated scalar multiplication for two different families of elliptic curves, namely Montgomery curves [27] and GLV curves [11]. In the former case, an SPA-protected scalar multiplication over a 160-bit OPF takes only $5.93 \cdot 10^6$ cycles, which is faster than most unprotected implementations reported in the literature. On the other hand, we use the GLV curve to explore the "lower bound" of the execution time for a scalar multiplication when resistance against SPA is not needed. Such a speed-optimized implementation has an execution time of only $4.19 \cdot 10^6$ clock cycles for a 160-bit scalar.

2 Preliminaries

In this section we recap some basic properties of special families of prime fields and elliptic curves, and discuss how to exploit their distinctive features to speed up the arithmetic operations needed in ECC.

2.1 Prime Fields

Even though elliptic curves can be defined over various algebraic structures, we only consider prime fields in this paper [6]. Formally, a prime field $\mathbb{F}_p$ consists of p elements (namely the integers between 0 and $p-1$) and the arithmetic operations are addition and multiplication modulo p. It is common practice in ECC to use "special" primes to speed up the modular reduction; a well-known example for primes with good arithmetic properties are the so-called Mersenne primes, which are primes of the form $p = 2^k - 1$. Multiplying two k-bit integers $a, b \in \mathbb{F}_p$ yields a $2k$-bit product r that can be written $r = r_H \cdot 2^k + r_L$, where r_H and r_L represent the upper half and the lower half of r, respectively. Since $2^k \equiv 1 \bmod p$, we can simply reduce r via a conventional addition of the form $t = (r_H + r_L) \bmod p$ to obtain a result that is at most $k+1$ bits long. A final subtraction of p may be necessary to get a fully reduced result. In summary, a reduction modulo a Mersenne prime requires just a conventional k-bit addition and, in the worst case, a subtraction of p. Unfortunately, Mersenne primes are rare, and there exist no Mersenne primes between 2^{160} and 2^{512}, which is the interval from which one normally chooses primes for ECC.

A wealth of research has been devoted to find other families of prime fields that allow for similarly efficient arithmetic and many proposals appeared in the literature, e.g. fields based on "Mersenne-like" primes such as pseudo-Mersenne primes [7] and generalized Mersenne primes [33]. A *pseudo-Mersenne prime* is a prime of the form

$$p = 2^k - c \tag{1}$$

where $\log_2(c) \leq \frac{1}{2}k$, i.e. the constant c is small compared to 2^k. However, c is bigger than 1, and hence the reduction operation modulo such a prime is more costly than that for a "real" Mersenne prime. On the other hand, allowing c to be bigger than 1 provides a larger choice of primes for a given bit-length.

The so-called *generalized Mersenne primes* were first described by Solinas in 1999 [33] and shortly thereafter, the NIST recommended a set of five of these special primes for use in ECC cryptosystems. The common form of the primes presented by Solinas is

$$p = 2^k - c_1 2^{k-1} - \cdots - c_i 2^{k-i} - \cdots - c_k \tag{2}$$

where all c_i are integers with a small absolute value, e.g. $c_i \in \{-1, 0, 1\}$. A concrete example is $p = 2^{192} - 2^{64} - 1$, which is one of the primes recommended by the NIST. The reduction operation modulo generalized Mersenne primes is similar to that of real Mersenne primes, namely to exploit congruence relations that stem from the special form of the prime to "shorten" the residue.

2.2 Elliptic Curves

Any elliptic curve over a prime field $\mathbb{F}_p$ can be expressed through a Weierstrass equation of the form $y^2 = x^3 + ax + b$ [16]. When using mixed Jacobian-affine coordinates, a point addition on a Weierstrass curve costs eight multiplications (i.e. 8 M) and three squarings (i.e. 3 S) in the underlying field, whereas a point doubling requires 4 M and 4 S [16]. Similar to prime fields, there exist numerous "special" families of elliptic curves, each having a unique curve equation and a unique addition law. In the past 20 years, a massive research effort was devoted to finding special curves that allow for a more efficient implementation of the scalar multiplication than ordinary Weierstrass curves.

Peter Montgomery introduced in 1997 a family of curves to speed up algorithms for the factorization of big integers [27]. These curves are referred to as Montgomery curves and have the unique property that a scalar multiplication can be carried out using the x coordinate only, which is much faster than when both the x and y coordinate are calculated in each step [6]. In formal terms, a Montgomery curve over $\mathbb{F}_p$ is defined by the equation

$$By^2 = x^3 + Ax^2 + x \tag{3}$$

with $A, B \in \mathbb{F}_p$, $(A^2 - 4)B \neq 0$ and allows for a very fast computation of the x-coordinate of the sum $P + Q$ of two points P, Q whose difference $P - Q$ is known. A point addition performed via the equation from [27, p. 261] requires 4 M and 2 S, whereas a point doubling costs 3 M and 2 S. However, one of the three field multiplications in the point doubling uses the constant $(A + 2)/4$ as operand, which is small if the parameter A is chosen accordingly. Our results show that multiplying a field element by a small constant (up to 16 bits) costs only between 0.2 M and 0.25 M (cf. Section 4). Furthermore, the point addition formula given in [27, p. 261] can be optimized when using the so-called Montgomery ladder (Alg. 13.35 in [6]) for scalar multiplication and representing the base point in projective coordinates with $Z = 1$ (see also Remark 13.36 (ii) in [6]). Even though the number of field multiplications and squarings is low, one has to take into account that the Montgomery ladder always executes both a point addition and a point doubling for each bit of the scalar k. Therefore, the computational cost of a scalar multiplication amounts to $5.25n$ multiplications and $4n$ squarings in $\mathbb{F}_p$, i.e. 5.25 M + 4 S per bit.

The so-called Gallant-Lambert-Vanstone curves, or simply GLV curves, are elliptic curves over $\mathbb{F}_p$ which possess an efficiently computable endomorphism ϕ whose characteristic polynomial has small coefficients [11]. The specific curve we use in this paper belongs to the family of GLV curves that can be described by a Weierstrass equation of the form

$$y^2 = x^3 + b \quad \text{(i.e. } a = 0 \text{ and } b \neq 0\text{)} \tag{4}$$

over a prime field $\mathbb{F}_p$ with $p \equiv 1 \bmod 3$ (see Example 4 from [11]). When using mixed Jacobian-affine coordinates, the point addition on such a curve requires 8 M + 3 S, i.e. adding points is exactly as costly as on an ordinary Weierstrass

curve. On the other hand, the double of a point given in Jacobian coordinates can be computed with only 3 M + 4 S since the parameter a of our GLV curve is 0. However, what makes GLV curves really attractive is that the cost for the computation of a scalar multiplication can be significantly reduced thanks to an efficiently-computable endomorphism as described in [11]. This endomorphism allows one to accomplish an n-bit scalar multiplication $k \cdot P$ by a computation of the form $k_1 \cdot P + k_2 \cdot Q$, whereby k_1, k_2 have only half the length of k. The two half-length scalar multiplications can be carried out simultaneously (via "Shamir's trick"), which takes $n/2$ point doublings and roughly $n/4$ additions when k_1, k_2 are represented in Joint Sparse Form (JSF) [16]. Thus, the overall cost of computing $k \cdot P$ amounts to $3.5n$ multiplications and $2.75n$ squarings in $\mathbb{F}_p$, i.e. 3.5 M + 2.75 S per bit.

3 Optimal Prime Fields

The lightweight ECC software we introduce in this paper is based on a special family of prime fields, the so-called *Optimal Prime Fields (OPFs)*, which were first described in the literature in an extended abstract from 2006 [12]. OPFs are defined by "low-weight" primes that can be written as

$$p = u \cdot 2^k + v \tag{5}$$

where u and v are small compared to 2^k, e.g. have a length of 8 or 16 bits so that they fit into one two registers of an 8-bit processor. A concrete example is $p = 65356 \cdot 2^{144} + 1$ (i.e. $u = 65356$, $k = 144$, and $v = 1$), which happens to be a 160-bit prime that looks as follows when written as a hex-string:

$$p = \text{0xFF4C0000000000000000000000000000000000001}$$

The main characteristic of these primes is their low Hamming weight, which is due to the fact that only a few bits near to the Most Significant Bit (MSB) and the Least Significant Bit (LSB) are non-zero; all the "middle" bits in between are 0. This property distinguishes them from other families of primes used in ECC, in particular Mersenne-like primes (cf. Section 2.1), which generally have a high Hamming weight. Using primes with a low Hamming weight allows for a simplification of the modular multiplication and other operations since all the zero-bits (resp. zero-bytes) do not need to be processed in a reduction modulo p. Most modular reduction algorithms, including Barrett and Montgomery reduction [26], can be optimized for OPFs, as will be shown in more detail in the remainder of this section. Another advantage of OPFs is that there exist a large number of primes of the form $p = u \cdot 2^k + v$ for any bitlength, which is not the case for generalized Mersenne primes.

The implementation of most of the arithmetic operations we describe in the following subsections is based on Zhang et al's OPF library for AVR processors [43]. However, Zhang's library, in its original form, is not resistant against side-channel attacks because it contains operand-dependent conditional statements

such as if-then-else constructs. Therefore, we modified the arithmetic functions in a way so that they exhibit a highly regular execution pattern (and constant execution time) regardless of the actual values of the operands. In addition, we optimized a number of performance-critical code sections in the field arithmetic operations, which improved their execution time by up to 10% versus Zhang's OPF library. As stated in Section 1.2, we strive for a scalable implementation able to process operands of varying length. To achieve this, we implemented all arithmetic functions to support the passing of a length parameter, which is then used by the function to calculate the number of loop iterations. Our library is dimensioned for operands between 64 and 2048 bits in steps of 32 bits, i.e. the operand length has to be a multiple of 32.

3.1 Selection of Primes

The original definition of OPFs in [12] specifies the coefficients u and v of the prime $p = u \cdot 2^k + v$ to "fit into a single register of the target processor," i.e. in our case, u and v would be (at most) 8 bits long. However, the OPF library we describe in this paper expects u to be a 16-bit integer, while v is fixed to 1. In the following, we explain the rationale behind this choice and elaborate on the supported bitlengths of p.

It is common practice in ECC to use primes with a bitlength that is a multiple of 32, e.g. 160, 192, 224 and 256 bits for applications with low to medium security requirements, and 384 and 512 bits for high-security applications. All standardization bodies (e.g. NIST, IEEE, SECG) recommend primes of these lengths and also we follow this approach. However, for efficiency reasons, it can be advantageous to use finite fields of a length slightly smaller than a multiple of 32, e.g. 255 bits instead of 256 [4]. Such slightly reduced field sizes facilitate certain optimization techniques like the so-called "lazy reduction," which means that the result of an addition or any other operation is only reduced when it is necessary so as to prevent overflow. We conducted some experiments with the 159-bit OPF given by $p = 126 \cdot 2^{152} + 1$, but found the performance gain one can achieve through lazy reduction to be less than 5%. Therefore, we decided to stick with the well-established field lengths of 160, 192, 224 and 256 bits.

Our OPF software uses Montgomery's algorithm [26] for multiplication and squaring modulo p. A standard implementation of Montgomery multiplication based on e.g. the so-called Finely Integrated Product Scanning (FIPS) method [21] has to execute $2s^2 + s$ word-level multiplications (i.e. $(w \times w)$-bit `mul` instructions) for operands consisting of s words [14]. However, when we optimize the FIPS method for primes of the form $p = u \cdot 2^k + v$ with $0 < u, v < 2^w$, then only $s^2 + 3s$ `mul` instructions are required since all the "middle" words of p do not need to be processed because they are 0. A further reduction is achievable if $v = 1$ since this case simplifies the quotient determination in Montgomery's algorithm so that only $s^2 + s$ `mul` instructions need to be executed, as we will show in Section 3.3. The situation is similar for $v = 2^w - 1$ (which corresponds to $v = -1$ in two's complement representation) as also this special case allows for a reduction of the number of `mul` instructions. Having $v = 2^w - 1$ implies

that the least significant word of p is an "all-one" word, which, in turn, means $p \equiv 3 \bmod 4$ and square roots modulo p can be computed efficiently [6].

The bitlength of a prime of the form $p = u \cdot 2^k + 1$ is not only determined by the exponent k, but also the coefficient u. To maximize performance, it was recommended in [12] to select u so that its length matches the word-size of the underlying processor; in our case, u should be an 8-bit integer in order to fit in a single register of an ATmega128 processor. When doing so, an optimized FIPS Montgomery multiplication ignoring all the zero-bytes of p requires to execute only $s^2 + s$ `mul` instructions. However, high performance is only one of several design goals; as stated in Section 1.2, we also aim for scalability, which means the ability to support fields of different lengths without the need to re-compile the arithmetic library. Besides the common field lengths of 160, 192, 224, and 256 bits, we want our library also to be able to perform arithmetic in 384 and 512-bit OPFs. Unfortunately, neither a 384-bit nor a 512-bit prime of the form $p = u \cdot 2^k + 1$ with $2^7 <= u < 2^8$ exists. It should be noted that the situation is very similar for pseudo-Mersenne primes; none of the 256 integers of the form $2^k - c$ with $k = 384$ and $c < 2^8$ is prime, and the same holds for $k = 512$. As a consequence, we decided to "weaken" the original criterium for the selection of u, namely to fit into a single register on the target processor, and allow u to have a length of 16 bits. While this relaxed condition for the selection of u entails a slight performance degradation, it significantly increases scalability and allows our OPF library to support high-security applications requiring 384 and 512-bit fields. All arithmetic functions of our library assume that u is a 16-bit integer and can be kept in two registers of an 8-bit ATmega128 processor. The second coefficient v of our low-weight primes is fixed to 1.

Notation. In what follows, $\mathbb{F}_p$ denotes an OPF defined by a prime of the form $p = u \cdot 2^k + 1$, whereby u is in the range $[2^{15}, 2^{16} - 1]$, i.e. u has a length of 16 bits. As mentioned above, the bitlength n of the primes we use in this paper is always a multiple of 32, e.g. $n = 160$, 192, 224, or 256 bits. Field elements are referred to by lowercase italic letters, e.g. $a \in \mathbb{F}_p$. When implementing ECC in software, it is common practice to represent field elements by arrays of single-precision (i.e. w-bit) words so that the arithmetic operations can be executed efficiently on the processor's fast integer unit [16]. Normally, one chooses w to match the word-size of the underlying processor, which would mean $w = 8$ in the case of an 8-bit processor. However, as shown by Gura et al in [15], it can be more efficient to process several (e.g. four) bytes of the operands at a time (instead of just a single byte), which, in fact, means to work with 32-bit words even though the processor has just an 8-bit datapath. We follow this approach and represent the elements of $\mathbb{F}_p$ via arrays of $s = \lceil n/w \rceil$ words, each having a length of $w = 32$ bits. For example, an element of a 160-bit prime field consists of five 32-bit words since $s = 160/32 = 5$. We use uppercase letters to denote these arrays and indexed uppercase letters to refer to individual words within an array, e.g. $A = (A_{s-1}, \ldots, A_1, A_0)$ where A_0 is the least significant word and A_{s-1} the most significant word of A, respectively.

3.2 Modular Addition and Subtraction

The typical way to perform a modular addition $z = a + b \bmod p$ is to first add the two n-bit operands $a, b \in \mathbb{F}_p$ to get a temporary sum $t = a + b$ (which can have a length of up to $n + 1$ bits), followed by a comparison between t and p to check whether $t \geq p$. Based on the result of this comparison, it may be necessary to subtract p from t to get a sum in the range of $[0, p - 1]$. However, this approach exhibits an operand-dependent (and, therefore, irregular) execution pattern that leaks information through small variations of both the execution time and power consumption profile, the latter of which may be exploited in an SPA attack as described in e.g. [34]. In fact, this side-channel leakage has two origins, one is the comparison between t and p, and the other is the conditional subtraction of p. Most performance-optimized ECC implementations adopt an "early-abort" strategy to compare two integers, which means the comparison is done word by word, starting at the most significant word-pair, and the result is immediately returned when the first unequal word-pair is found. Therefore, the difference between the operands determines the execution time; it is maximal when the operands are equal. The second origin of side-channel leakage, i.e. the subtraction of p, is more obvious since this subtraction is only performed when the temporary sum t is not smaller than p.

In order to eliminate or, at least, reduce side-channel leakage, we adopt the idea of incomplete modular arithmetic as described by Yanık et al [42]. Instead of reducing the result t of the addition to the least non-negative residue in the range of $[0, p - 1]$, incomplete modular arithmetic allows (i.e. tolerates) results that are not fully reduced as long as they do not exceed a certain bitlength. In our case, this means that all results of modular operations are (at most) n bits long, but do not necessarily need to be smaller than p. All our modular arithmetic functions also accept incompletely reduced operands as inputs, provided that their length does not exceed n, the bitlength of p. The advantage of this "relaxed" residue representation is the possibility to perform modular addition without an exact comparison between the sum t and the prime p. Instead, we just check whether the length of t exceeds n bits (i.e. whether $t \geq 2^n$), which is only the case when the addition $t = a + b$ produced a "carry bit."

Thanks to the carry bit (which is either 0 or 1), the conditional subtraction of p can be done in an "unconditional" way by applying a mask to each byte of p before it is subtracted. The value of this mask is either an "all-zero" byte or an "all-one" byte and can be easily obtained from the carry bit through negation. For example, when the carry bit $c = 0$, the value of the mask becomes $m = -c = 0$. Applying this mask m to a byte p_i of p (i.e. performing a logical `and` between m and p_i) yields a zero-byte, which means 0 is subtracted from the sum t. Conversely, when $c = 1$, we have $m = -c = -1 = 2^8 - 1 = 0$xff, and applying this m to the bytes p_i does not change their value, which means p is subtracted from t. Note, however, that a second subtraction may be required to obtain an n-bit result since both operands can be incompletely reduced. To get "branch-less" code, we always perform two masked subtractions of p and update the carry bit c after the first one. More precisely, the first subtraction

produces a "borrow bit," which is either 0 or 1 and has to be subtracted from the carry bit to obtain the correct carry bit for the second subtraction.

A modular subtraction $z = a - b \bmod p$ can be implemented on basis of the same principles as the modular addition described above. Our implementation performs an ordinary subtraction $t = a - b$ followed by two masked additions of p, whereby the mask is derived from the borrow bit of the subtraction.

3.3 Modular Multiplication and Squaring

As detailed earlier in this section, our OPF library supports low-weight primes of the form $p = u \cdot 2^k + 1$ where u is 16 bits long. Following the notation from Section 3.1, we can represent p via an array $P = (P_{s-1}, \ldots, P_1, P_0)$ consisting of s words, each having a length of w bits, i.e. $w/4$ bytes. The least significant word P_0 is 1, while the most significant word P_{s-1} contains u; all other words are 0. In this subsection, we assume that the two operands a, b to be multiplied have the same length as p, namely n bits, but they do not necessarily need to be smaller than p, i.e. a and b are in the range of $[0, 2^n - 1]$.

We show in the following that Montgomery modular multiplication [26] can be optimized for primes of the form $p = u \cdot 2^k + 1$ by simply ignoring all words P_i with $1 \leq i \leq s - 2$ (i.e. all "zero" words) in the reduction operation. When doing so, the overall number of word-level (i.e. $(w \times w)$-bit) multiplications to compute a Montgomery product amounts to $s^2 + s$, of which s^2 contribute to the multiplication of the s-word operand a by b, and the rest to the reduction modulo p. In other words, the "overhead" of modular reduction is only s word-level multiplications, i.e. reduction has linear complexity. For comparison, the reduction of a $2s$-word product modulo a pseudo-Mersenne prime of the form $p = 2^k - c$ (with c fitting into a single word) also requires exactly s word-level multiplications [6]. However, when performing a modular multiplication with a pseudo-Mersenne prime, the reduction is typically done *after* the multiplication (see e.g. [18]), which is inefficient since the $2s$-word product is first written to memory (during the multiplication), and then it has to be loaded again from memory to accomplish the reduction. To avoid this, our implementation adopts a variant of the so-called Finely Integrated Product Scanning (FIPS) method [21] for Montgomery multiplication, which interleaves multiplication steps and reduction steps instead of executing them one after the other, thereby saving a number of load/store instructions and reducing the RAM footprint.

The standard FIPS technique for arbitrary primes, as described in [21] and [14], has a nested-loop structure with two outer and two simple inner loops. In each iteration of the inner loops, two Multiply-Accumulate (MAC) operations are carried out; one with the words of the operands a and b, which contributes to the computation of $a \cdot b$. The second MAC operation involves words of the prime p and, hence, contributes to the reduction operation. Algorithm 1 shows a special variant of the FIPS method optimized for "low-weight" primes of the form $p = u \cdot 2^k + 1$. This variant differs from the generic FIPS method for arbitrary primes in three main aspects. First, we eliminated all multiplications and MAC operations performed on zero words of p since they do not contribute to

Algorithm 1. FIPS Montgomery modular multiplication for OPFs

Input: An n-bit prime $p = u \cdot 2^k + 1$ given as s-word array $P = (P_{s-1}, \ldots, P_1, P_0)$, two integers $a, b \in [0, 2^n - 1]$ given as $A = (A_{s-1}, \ldots, A_1, A_0)$, $B = (B_{s-1}, \ldots, B_1, B_0)$.
Output: An $(s+1)$-word array $Z = (Z_s, \ldots, Z_1, Z_0)$ with $Z_s \in \{0, 1\}$ representing a possibly incompletely reduced Montgomery product $z = a \cdot b \cdot 2^{-n} \bmod p$.

1: $T \leftarrow A_0 \times B_0$
2: **for** i from 1 by 1 to $s - 1$ **do**
3: $\quad Z_{i-1} \leftarrow -(T \bmod 2^w)$
4: $\quad T \leftarrow T + Z_{i-1}$
5: $\quad T \leftarrow T / 2^w$
6: $\quad$ **for** j from 0 by 1 to i **do**
7: $\quad\quad T \leftarrow T + A_j \times B_{i-j}$
8: $\quad$ **end for**
9: **end for**
10: $T \leftarrow T + Z_0 \times P_{s-1}$
11: $Z_{s-1} \leftarrow -(T \bmod 2^w)$
12: $T \leftarrow T + Z_{s-1}$
13: **for** i from s by 1 to $2s - 2$ **do**
14: $\quad T \leftarrow T / 2^w$
15: $\quad$ **for** j from $i - s + 1$ by 1 to $s - 1$ **do**
16: $\quad\quad T \leftarrow T + A_j \times B_{i-j}$
17: $\quad$ **end for**
18: $\quad T \leftarrow T + Z_{i-s+1} \times P_{s-1}$
19: $\quad Z_{i-s} \leftarrow T \bmod 2^w$
20: **end for**
21: $T \leftarrow T / 2^w$
22: $Z_{s-1} \leftarrow T \bmod 2^w$
23: $Z_s \leftarrow T / 2^w \quad \{ Z_s \text{ is either 0 or 1} \}$
24: $Z \leftarrow (Z_s, \ldots, Z_1, Z_0)$
25: **return** Z

the final result. Consequently, the inner loops of Algorithm 1 perform only one MAC operation, similar to the product-scanning method for multiple-precision multiplication [16]. In fact, the inner loops in line 6–8 and 15–17 are the same as in product-scanning multiplication, which makes Algorithm 1 fairly easy to implement. Another difference between our FIPS variant and the generic FIPS method for arbitrary primes is that the former is optimized for $P_0 = 1$ and, as a consequence, the Montgomery reduction requires only s MAC operations; one is performed in line 10 and the remaining $s - 1$ in the second outer loop (line 18). When $P_0 = 1$, we have $-P_0^{-1} \bmod 2^w = -1 \bmod 2^w = 2^w - 1$, which simplifies the quotient-determination part of the reduction operation compared to the original FIPS method (see [43, Section 4.3] for a detailed explanation). Due to this optimization, the total number of word-level multiplications and MAC operations of the FIPS method for $p = u \cdot 2^k + 1$ amounts to only $s^2 + s$. The third difference between our FIPS variant and the classic one is that we peeled off the computation of $A_0 \times B_0$ from the first nested loop and re-arranged the

loop structure accordingly. Because of this modification, all loops of Algorithm 1 iterate at least one time if $s \geq 2$, which simplifies their implementation.

Our AVR Assembly implementation of the FIPS Montgomery multiplication is based on the pseudo-code from Algorithm 1. However, in order to maximize performance, we adopt a variant of Gura et al's hybrid multiplication method [15], which means all word-level multiplications and MAC operations are performed on four bytes (i.e. 32 bits) of the operands instead of just a single byte (i.e. our word-size w is 32). In each iteration of the two inner loops, four bytes of operand a (i.e. the word A_j) and operand b (i.e. the word B_{i-j}) are loaded from memory and multiplied together to a 64-bit product. This product is then added to a cumulative sum T held in nine 8-bit registers. Our implementation of the inner loops follows [24, Section 3.1] and is, therefore, slightly faster than Zhang et al's inner-loop operation from [43]. Each iteration of the inner loops consists of eight `ld` (i.e. load), 16 `mul`, 49 `add` (or `adc`), and four `movw` instructions (excluding loop overhead). When taking the updating of the loop-control variable and branch instruction into account, the overall execution time of one full iteration of the inner loop amounts to exactly 104 clock cycles.

Besides excellent performance, the inner-loop implementation from [24] has the further advantage that it occupiers only 30 out of the 32 working registers of an AVR processor. We use the two free registers to accommodate the 16-bit coefficient u of the prime $p = u \cdot 2^k + 1$. Hence, we have to maintain only three pointers, namely the pointers to the arrays A, B, and Z, which we hold in the three pointer registers `X`, `Y`, and `Z` during the execution of a multiplication. In each iteration of the inner loop, the pointer to A gets incremented by 4, while the pointer to B is decremented. Therefore, the pointers need to be initialized with the correct start addresses, and this initialization has to performed in the outer loop, immediately before the start of the inner loop. Zhang et al [43] did this pointer initialization with help of the "original" start address of the arrays A and B (i.e. the address of A_0 and B_0), which they pushed on the stack at the very beginning of the multiplication and then popped whenever needed. Unfortunately, this approach is quite expensive since `push` and `pop` instructions take two cycles each. We found it more efficient to re-calculate the original address of these pointers using the end-value of the loop counter.

Algorithm 1 does not include the so-called "final subtraction" of p, which is generally required in Montgomery multiplication to guarantee that the result is smaller than p or, in our case, smaller than 2^n. Therefore, the array Z consists of $s + 1$ words, whereby its most significant word Z_s is either 0 or 1. Note that (at most) one subtraction of p is required to get an s-word result in the range of $[0, 2^n - 1]$, even when both inputs are not completely reduced. To minimize SPA leakage, we perform this subtraction of p in the same way as described in Section 3.2, but use Z_s to derive an "all-zero" or "all-one" mask.

We implemented modular squaring for our low-weight primes similar to the multiplication, using the same optimizations in the reduction. Furthermore, the squaring adopts the well-known "trick" that allows one to cut the total number of word-level multiplications by almost one half (from s^2 to $\frac{s^2+s}{2}$) [6].

4 Performance Evaluation and Comparison

In the following, we present execution times of both field and group arithmetic operations, including scalar multiplication, for OPFs (and appropriate elliptic curves) ranging from 160 to 256 bits. As mentioned before, we implemented all OPF arithmetic operations in Assembly language to achieve peak performance on 8-bit AVR processors. The group operations (i.e. the point arithmetic) and the algorithms for scalar multiplication were written in ANSI C and compiled using WinAVR. We determined the execution time of all arithmetic operations with help of the cycle-accurate instruction-set simulator of AVR Studio 4.

Table 1. Execution time (in clock cycles) of arithmetic operations in OPFs

Operation	160 bit	192 bit	224 bit	256 bit
Addition	530	631	732	833
Subtraction	530	631	732	833
Multiplication	3237	4500	5971	7650
Squaring	2901	3909	5058	6347
Mul. by 16-bit integer	873	1039	1295	1461
Inversion	223374	311828	416758	531901

Table 1 summarizes the execution times we obtained using the ATmega128 processor as target platform, whereby all timings include the full function-call overhead. A multiplication in a 160-bit OPF takes 3237 clock cycles, which is almost 10% faster than the average multiplication time of 3542 cycles reported by Zhang et al [43]. For comparison, Szczechowiak et al's NanoECC [35] needs a total of 3882 clock cycles for a 160-bit modular multiplication (2654 cycles to do the multiplication, 1228 cycles for a reduction modulo a 160-bit generalized Mersenne prime), even though they fully unrolled the loops. The overhead due to the reduction operation accounts for about 31.6% of the total multiplication time. On the other hand, the reduction overhead of multiplication in a 160-bit OPF is 459 clock cycles (or 14.2%) since, according to [24], a conventional 160-bit multiplication (without modular reduction) requires 2778 cycles.

As analyzed in Section 3.3, our FIPS Montgomery multiplication for OPFs has to perform $s^2 + s$ word-level multiplications or MAC operations, which are essentially (32×32)-bit multiplications in our case. On an 8-bit processor, this translates into $16s^2 + 8s$ `mul` instructions since the two least significant bytes of P_{s-1} are 0, i.e. the s MAC operations in line 10 and 18 of Algorithm 1 need only eight `mul` instructions instead of 16. On the other hand, an OPF squaring including reduction involves $(s^2 + 3s)/2$ word-level multiplications (resp. MAC operations), which means $8s^2 + 12s$ `mul` instructions on the ATmega128. As a consequence, one would expect OPF squaring to be (almost) 50% faster than OPF multiplication. However, an optimized squaring function has to carry out some auxiliary operations, e.g. left-shifts of word-level (i.e. 64-bit) products in

Table 2. Execution time (in cycles) of point arithmetic and scalar multiplication

Operation	160 bit	192 bit	224 bit	256 bit
GLV point addition	40305	54417	70418	88550
GLV point doubling	26684	36539	45369	56296
GLV scalar mul.	4191073	6918518	10064582	14178625
Montgomery point add.	19479	25890	33207	41428
Montgomery point dbl.	15950	21072	26884	33390
Montgomery scalar mul.	5928088	9445554	14109549	20158840

order to double them, which significantly impacts the total execution time, the more so the shorter the operands are. The results given in Table 1 show that, in a 160-bit OPF, squaring is just some 10.4% faster than multiplication, but the gain increases to roughly 17% in a 256-bit OPF. We implemented the inversion based on the binary version of the well-known Extended Euclidean Algorithm (EEA) [16], whereby we exploited the special form of our primes to accelerate certain low-level operations, e.g. additions and subtractions of p. Note that the inversion has an irregular execution profile, which means the execution time is not constant but depends on the input. Table 1 specifies the average execution time of 100 inversions performed on random field elements.

Table 2 lists the simulated execution times of point addition/doubling and full scalar multiplication for both GLV and Montgomery curves. As explained in Section 2.2, the addition and doubling of points on a Montgomery curve is less costly (in terms of arithmetic operations in the underlying prime field) than the point addition/doubling on a GLV curve, and the simulation results from Table 2 clearly confirm this. However, the situation becomes different when we compare the execution times of a full scalar multiplication since the GLV curve outperforms its Montgomery counterpart by a factor of 1.41 in the 160-bit case (i.e. $4.19 \cdot 10^6$ versus $5.93 \cdot 10^6$ cycles on an ATmega128). We implemented the scalar multiplication on the Montgomery curve in a straightforward way based on a "Montgomery ladder" [6], while the scalar multiplication on the GLV curve exploits an efficiently computable endomorphism as described in [11, 16]. Since the Montgomery curves we used have a positive trace and a co-factor of 4, we evaluated the execution time using scalars that are two bits shorter than the underlying OPF. On the other hand, our GLV curves have a co-factor of 1 and we used scalars k that satisfy the following conditions: (1) the two sub-scalars k_1, k_2 of the de-composition of k are both positive and $n/2$ bits long (n is the bitlength of the underlying OPF), and (2) their JSF contains $n/4$ zero bits.

Table 3 compares the scalar multiplication time of our two implementations with previous results reported in the literature. Our GLV variant outperforms all previous implementations, with two exceptions, namely the implementation of Aranha et al [1] and Wenger et al [40]. However, both applied extensive loop unrolling in the field arithmetic operations, which in general entails large code size and poor scalability. Furthermore, the implementation of Aranha et al can only be made SPA resistant at the expense of a massive performance hit.

Table 3. Comparison of execution time of scalar multiplication over fields of an order of roughly 160 bits (evaluation platform is an ATmega128 clocked at 7.3728 MHz)

Implementation	Field order	Fixed P.	Rand. P.	SPA resistant
Seo et al [32]	GF(2^m), 163 bit	1.14 s	1.14 s	No
Kargl et al [20]	GF(2^m), 167 bit	0.76 s	0.76 s	No
Aranha et al [1]	GF(2^m), 163 bit	0.29 s	0.32 s	No
Liu et al [23]	GF(p), 160 bit	2.05 s	2.30 s	No
Szczechowiak et al [35]	GF(p), 160 bit	1.27 s	1.27 s	No
Wang et al [39]	GF(p), 160 bit	1.24 s	1.35 s	No
Gura et al [15]	GF(p), 160 bit	0.88 s	0.88 s	No
Chu et al [5]	GF(p), 160 bit	0.79 s	0.79 s	No
Großschädl et al [13]	GF(p), 160 bit	0.74 s	0.74 s	No
Ugus et al [36]	GF(p), 160 bit	0.57 s	1.03 s	No
Wenger et al [40] (Mon.)	GF(p), 160 bit	0.75 s	0.75 s	Yes
Wenger et al [40] (GLV)	GF(p), 160 bit	0.53 s	0.53 s	No
Our work (Montg. curve)	GF(p), 160 bit	0.80 s	0.80 s	Yes
Our work (GLV curve)	GF(p), 160 bit	0.57 s	0.57 s	No

5 Conclusions

The aim of this paper was to provide new insights into certain implementation aspects of OPFs on 8-bit AVR processors. First, we argued that OPFs defined by primes of the form $p = u \cdot 2^k + 1$, where u is a 16-bit integer, represent an optimal trade-off between performance and scalability. Then, we described in detail how to implement arithmetic operations for OPFs, taking the properties (e.g. low Hamming weight) of these primes into account. In particular, we proposed a new variant of Montgomery multiplication for low-weight primes based on the FIPS method. Our Montgomery variant has the same loop structure as the ordinary product-scanning method for multiplication and can, therefore, be well optimized for ATmega processors. We implemented the multiplication and and all other arithmetic operations needed for ECC in a parameterized fashion with rolled loops so as to achieve high scalability and small code size. Furthermore, we wrote the Assembly code of all arithmetic functions (bar inversion) in such a way that always the same instruction sequence is executed, irrespective of the actual value of the operands, which helps to foil SPA attacks. Simulation results obtained with AVR Studio 4 indicate an execution time of 3237 cycles for a multiplication in a 160-bit OPF, while squaring takes 2901 cycles. These results compare very favorably with previous work and outperform even some implementations with unrolled loops. We also evaluated the execution time of a full scalar multiplication on Montgomery as well as GLV curves over OPFs. In the former case, the scalar multiplication is "intrinsically" SPA resistant and executes in 5.93 million cycles over a 160-bit OPF, while, in the latter case, we have an execution time of 4.19 million cycles. Both results confirm that OPFs are an excellent implementation option for ECC on 8-bit AVR processors.

References

1. D. F. Aranha, R. Dahab, J. C. López, and L. B. Oliveira. Efficient implementation of elliptic curve cryptography in wireless sensors. *Advances in Mathematics of Communications*, 4(2):169–187, May 2010.
2. Atmel Corporation. 8-bit ARV® Instruction Set. User Guide, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, July 2008.
3. Atmel Corporation. 8-bit ARV® Microcontroller with 128K Bytes In-System Programmable Flash: ATmega128, ATmega128L. Datasheet, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, June 2008.
4. D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography — PKC 2006*, vol. 3958 of *Lecture Notes in Computer Science*, pp. 207–228. Springer Verlag, 2006.
5. D. Chu, J. Großschädl, Z. Liu, V. Müller, and Y. Zhang. Twisted Edwards-form elliptic curve cryptography for 8-bit AVR-based sensor nodes. In *Proceedings of the 1st ACM Workshop on Asia Public-Key Cryptography (AsiaPKC 2013)*, pp. 39–44. ACM Press, 2013.
6. H. Cohen and G. Frey. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, vol. 34 of *Discrete Mathematics and Its Applications*. Chapmann & Hall\CRC, 2006.
7. R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent No. 5,159,632, Oct. 1992.
8. 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.
9. G. de Meulenaer and F.-X. Standaert. Stealthy compromise of wireless sensor nodes with power analysis attacks. In *Mobile Lightweight Wireless Systems — MOBILIGHT 2010*, vol. 45 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 229–242. Springer Verlag, 2010.
10. T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni, F.-X. Standaert, and L. van Oldeneel tot Oldenzeel. Compact implementation and performance evaluation of block ciphers in ATtiny devices. In *Progress in Cryptology — AFRICACRYPT 2012*, vol. 7374 of *Lecture Notes in Computer Science*, pp. 172–187. Springer Verlag, 2012.
11. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphism. In *Advances in Cryptology — CRYPTO 2001*, vol. 2139 of *Lecture Notes in Computer Science*, pp. 190–200. Springer Verlag, 2001.
12. 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 Press, 2006.
13. J. Großschädl, M. Hudler, M. Koschuch, M. Krüger, and A. Szekely. Smart elliptic curve cryptography for smart dust. In *Quality of Service in Heterogeneous Networks — QSHINE 2010*, vol. 74 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 548–559. Springer Verlag, 2010.

14. 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.
15. 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.
16. D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.
17. S. Heyse, I. von Maurich, A. Wild, C. Reuber, J. Rave, T. Poeppelmann, and C. Paar. Evaluation of SHA-3 candidates for 8-bit embedded processors. Presentation at the 2nd SHA-3 Candidate Conference, Santa Barbara, CA, USA, Aug. 2010. Available for download at `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/`.
18. M. Hutter and P. Schwabe. NaCl on 8-bit AVR microcontrollers. In *Progress in Cryptology — AFRICACRYPT 2013*, vol. 7918 of *Lecture Notes in Computer Science*, pp. 156–172. Springer Verlag, 2013.
19. 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.
20. A. Kargl, S. Pyka, and H. Seuschek. Fast arithmetic on ATmega128 for elliptic curve cryptography. Cryptology ePrint Archive, Report 2008/442, 2008. Available for download at `http://eprint.iacr.org`.
21. Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
22. 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.
23. 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.
24. Z. Liu and J. Großschädl. New speed records for Montgomery modular multiplication on 8-bit AVR microcontrollers. Cryptology ePrint Archive, Report 2013/882, 2013. Available for download at `http://eprint.iacr.org`.
25. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Verlag, 2007.
26. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
27. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, Jan. 1987.
28. E. Oswald. Enhancing simple power-analysis attacks on elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems — CHES 2002*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 82–97. Springer Verlag, 2002.
29. Y. Sakai and K. Sakurai. Simple power analysis on fast modular reduction with NIST recommended elliptic curves. In *Information and Communications Security — ICICS 2005*, vol. 3783 of *Lecture Notes in Computer Science*, pp. 169–180. Springer Verlag, 2005.

30. M. Scott and P. Szczechowiak. Optimizing multiprecision multiplication for public key cryptography. Cryptology ePrint Archive, Report 2007/299, 2007. Available for download at `http://eprint.iacr.org`.
31. H. Seo and H. Kim. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In *Information Security Applications — WISA 2012*, vol. 7690 of *Lecture Notes in Computer Science*, pp. 55–67. Springer Verlag, 2012.
32. S. C. Seo, D.-G. Han, H. C. Kim, and S. Hong. TinyECCK: Efficient elliptic curve cryptography implementation over GF(2^m) on 8-bit Micaz mote. *IEICE Transactions on Information and Systems*, E91-D(5):1338–1347, May 2008.
33. J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR-99-39, Centre for Applied Cryptographic Research (CACR), University of Waterloo, Waterloo, Canada, 1999.
34. D. Stebila and N. Thériault. Unified point addition formulæ and side-channel attacks. In *Cryptographic Hardware and Embedded Systems — CHES 2006*, vol. 4249 of *Lecture Notes in Computer Science*, pp. 354–368. Springer Verlag, 2006.
35. P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab. NanoECC: Testing the limits of elliptic curve cryptography in sensor networks. In *Wireless Sensor Networks — EWSN 2008*, vol. 4913 of *Lecture Notes in Computer Science*, pp. 305–320. Springer Verlag, 2008.
36. O. Ugus, D. Westhoff, R. Laue, A. Shoufan, and S. A. Huss. Optimized implementation of elliptic curve based additive homomorphic encryption for wireless sensor networks. In *Proceedings of the 2nd Workshop on Embedded Systems Security (WESS 2007)*, pp. 11–16, 2007. Available for download at `http://arxiv.org/abs/0903.3900`.
37. 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.
38. C. D. Walter. Simple power analysis of unified code for ECC double and add. In *Cryptographic Hardware and Embedded Systems — CHES 2004*, vol. 3156 of *Lecture Notes in Computer Science*, pp. 191–204. Springer Verlag, 2004.
39. 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.
40. E. Wenger and J. Großschädl. An 8-bit AVR-based elliptic curve cryptographic RISC processor for the Internet of things. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture Workshops (MICROW 2012)*, pp. 39–46. IEEE Computer Society Press, 2012.
41. A. D. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *Smart Card Research and Advanced Applications*, vol. 180 of *International Federation for Information Processing*, pp. 71–92. Kluwer Academic Publishers, 2000.
42. T. Yanık, E. Savaş, and Ç. K. Koç. Incomplete reduction in modular arithmetic. *IEE Proceedings – Computers and Digital Techniques*, 149(2):46–52, Mar. 2002.
43. Y. Zhang and J. Großschädl. Efficient prime-field arithmetic for elliptic curve cryptography on wireless sensor nodes. In *Proceedings of the 1st International Conference on Computer Science and Network Technology (ICCSNT 2011)*, vol. 1, pp. 459–466. IEEE, 2011.