Advances in Mathematics of Communications
Volume 4, No. 2, 2010, 169–187
doi:10.3934/amc.2010.4.169

EFFICIENT IMPLEMENTATION OF ELLIPTIC CURVE CRYPTOGRAPHY IN WIRELESS SENSORS

Diego F. Aranha, Ricardo Dahab,
Julio López and Leonardo B. Oliveira

University of Campinas (UNICAMP)
Campinas - SP, CEP 13083-970, Brazil

(Communicated by Joan-Josep Climent)

Abstract. The deployment of cryptography in sensor networks is a challenging task, given the limited computational power and the resource-constrained nature of the sensoring devices. This paper presents the implementation of elliptic curve cryptography in the MICAz Mote, a popular sensor platform. We present optimization techniques for arithmetic in binary fields, including squaring, multiplication and modular reduction at two different security levels. Our implementation of field multiplication and modular reduction algorithms focuses on the reduction of memory accesses and appears as the fastest result for this platform. Finite field arithmetic was implemented in C and Assembly and elliptic curve arithmetic was implemented in Koblitz and generic binary curves. We illustrate the performance of our implementation with timings for key agreement and digital signature protocols. In particular, a key agreement can be computed in 0.40 seconds and a digital signature can be computed and verified in 1 second at the 163-bit security level. Our results strongly indicate that binary curves are the most efficient alternative for the implementation of elliptic curve cryptography in this platform.

1. Introduction

A Wireless Sensor Network (WSN) [5] is a wireless ad-hoc network consisting of resource-constrained sensoring devices (limited energy source, low communication bandwidth, small computational power) and one or more base stations. The base stations are more powerful and collect the data gathered by the sensor nodes so it can be analyzed. As any ad hoc network, routing is accomplished by the nodes themselves through hop-by-hop forwarding of data. Common WSN applications range from battlefield reconnaissance and emergency rescue operations to surveillance and environmental protection.

WSNs may be organized in different ways. In flat WSNs, all nodes play similar roles in sensing, data processing, and routing. In hierarchical WSNs, on the other hand, the network is typically organized into clusters, with ordinary cluster members and the cluster heads playing different roles. While ordinary cluster members are responsible for sensing, the cluster heads are responsible for additional tasks such as collecting and processing the sensing data from their cluster members, and forwarding the results towards the base stations.

2000 *Mathematics Subject Classification:* Primary: 11-04; Secondary: 94A60.
Key words and phrases: Efficient software implementation, cryptographic engineering, elliptic curve cryptography, finite field arithmetic.

Besides the vulnerabilities already present in ad-hoc networks, WSNs pose additional challenges: the sensor nodes are commonly distributed on locations physically accessible to adversaries; and the resources available in a sensor node are more limited than those in a conventional ad hoc network node, thus traditional solutions are not adequate. For example, the fact that sensor nodes should be discardable and consequently have low cost makes the integration of anti-tampering measures on these devices difficult.

Conventional public key cryptography systems such as RSA and DSA are impractical in this scenario due to the low processing power of sensor nodes. Until recently, security services such as confidentiality, authentication and integrity were achieved exclusively by symmetric techniques [26, 13]. Nowadays, however, elliptic curve cryptography (ECC) [22, 14] has emerged as a promising alternative to traditional public key methods on WSNs [8], because of its lower processing and storage requirements. These features motivate the search for increasingly efficient algorithms and implementations of ECC for such devices. The usual target platform is the MICAz Mote [10], a node commonly used on real WSN deployments, whose main characteristics are the low availability of RAM memory and the high cost of memory instructions, memory addressing and bitwise shifts by arbitrary amounts.

This work proposes optimizations for implementing ECC over binary fields, improving its limits of performance and viability. Experimental results show that binary elliptic curves offer significant computational advantages over prime curves when implemented in WSNs. Note that this observation contradicts a common misconception that sensor nodes are not sufficiently equipped to compute elliptic curve arithmetic over binary fields in an efficient way [8, 4].

Our main contributions in this work are:

- *Efficient implementations of multiplication, squaring, modular reduction and inversion in* $\mathbb{F}_{2^{163}}$ *and* $\mathbb{F}_{2^{233}}$*:* optimized versions of known algorithms are presented, reducing the number of memory accesses to obtain performance gains. The new optimizations produce the fastest implementation of binary field arithmetic published for this platform;
- *Efficient implementation of elliptic curve cryptography:* point multiplication algorithms are implemented on Koblitz curves and generic binary curves. The time for a scalar multiplication of a random point in a binary curve is 61% faster than the best implementation so far [12] and 57% faster than the best implementation over a prime curve [7] at the 160-bit security level. We also present the first point multiplication timings at the 233-bit security level in this platform. Performance is illustrated by executions of key agreement and digital signature protocols.

The remaining sections of this paper are organized as follows. Related work is presented in Section 2 and elementary elliptic curve concepts are introduced in Section 3. The platform characteristics are presented in Section 4. Section 5 investigates efficient implementations of finite field arithmetic in the target platform while Section 6 investigates efficient elliptic curve arithmetic. Section 7 presents implementation results and Section 8 concludes the paper.

2. Related work

Cryptographic protocols are used to establish security services in WSNs. Key agreement is a fundamental protocol in this context because it can be used to negotiate cryptographic keys suitable for fast and energy-efficient symmetric algorithms.

One possible solution for key agreement in WSNs is the deployment of pairing-based protocols, such as TinyTate [23] and TinyPBC [25], with the added advantage of not requiring communication. Here instead we focus on the performance side and assume that a simple one-pass Elliptic Curve Diffie-Hellman [3] protocol is employed for key agreement. With this assumption, different implementations of ECC can be compared by the cost of multiplying a random elliptic point by a random integer.

Gura et al. [8] presented the first implementation results of ECC and RSA on ATmega128 microcontrollers and demonstrated the superiority of the former over the latter. In Gura's work, prime field arithmetic was implemented in C and Assembly and a point multiplication took 0.81 seconds on a 8MHz device. Uhsadel et al. [32] later presented an expected time of 0.76 seconds for computing a point multiplication in a 7.3728MHz device. The fastest implementation of prime curves so far [7] explores the potential of elliptic curves with efficient computable endomorphisms defined over optimal prime fields and computes a point multiplication in 5.5 million cycles, or 0.745 second.

For binary curves, Malan et al. [20] implemented ECC using polynomial basis and presented results for the Diffie-Hellman key agreement protocol. A public key generation, which consists of a point multiplication, was computed in 34 seconds. Yan and Shi [34] implemented ECC over $\mathbb{F}_{2^{163}}$ and obtained a point multiplication in 13.9 seconds, suggesting that binary curves had too high a cost for sensors' current technology. Eberle et al. [4] implemented ECC in Assembly over $\mathbb{F}_{2^{163}}$ and obtained a point multiplication in 4.14 seconds, making use of architectural extensions for additional acceleration. NanoECC [31] specialized portions of the MIRACL arithmetic library [28] in the C programming language for efficient execution in sensor nodes, resulting in a point multiplication in 2.16 seconds over prime fields and 1.27 seconds over binary fields. Later, TinyECCK [29] presented an implementation of ECC over binary curves which takes into account the platform characteristics to optimize finite field arithmetic and obtained a point multiplication in 1.14 second. Recently, Kargl et al. [12] investigated algorithms resistant to simple power analysis and obtained a point multiplication in 0.7633 second on a 8MHz device. Table 1 presents the increasing efficiency of ECC in WSNs.

Finite field	**Work**	**Execution time (seconds)**
Binary	Malan et al. [20]	34
	Yan and Shi [34]	13.9
	Eberle et al. [4]	4.14
	NanoECC [31]	2.16
	TinyECCK [29]	1.14
	Kargl et al. [12]	0.83
Prime	Wang and Li. [33]	1.35
	NanoECC [31]	1.27
	Gura et al. [8]	0.87
	Uhsadel et al. [32]	0.76
	TinySA [7]	0.745

Table 1. Timings for scalar multiplication of a random point on a MICAz Mote at the 160-bit security level. The timings are normalized for a clock frequency of 7.3728MHz.

3. Elliptic curve cryptography

An *elliptic curve* E over a field $\mathbb{K}$ is the set of solutions $(x, y) \in \mathbb{K} \times \mathbb{K}$ which satisfy the *Weierstrass equation*

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$ and the curve *discriminant* is $\Delta \neq 0$; together with a *point at infinity* denoted by $\mathcal{O}$. If $\mathbb{K}$ is a field of characteristic 2, then the curve is called a *binary elliptic curve* and there are two cases to consider. If $a_1 \neq 0$, then an admissible change of variables transforms E to the *non-supersingular binary elliptic curve* of equation

$$y^2 + xy = x^3 + ax^2 + b$$

where $a, b \in \mathbb{F}_{2^m}$ and $\Delta = b$. A non-supersingular curve with $a \in \{0, 1\}$ and $b = 1$ is also a *Koblitz curve*. If $a_1 = 0$, then an admissible change of variables transforms E to the *supersingular binary elliptic curve*

$$y^2 + cy = x^3 + ax + b$$

where $a, b, c \in \mathbb{F}_{2^m}$ and $\Delta = c^4$.

The number of points on the curve $E(\mathbb{F}_{2^m})$, denoted by $\#E(\mathbb{F}_{2^m})$, is called the *curve order* over the field $\mathbb{F}_{2^m}$. The *Hasse bound* enunciates in this case that $n = 2^m + 1 - t$ and $|t| \leq 2\sqrt{2^m}$, where t is the *trace of Frobenius*. A curve can be generated with a prescribed order using the complex multiplication method [15] or the curve order can be explicitly computed in binary curves using the approach due to Satoh, Skjernaa and Taguchi [27]. Non-supersingularity comes from the fact that t is not a multiple of the characteristic 2 of the underlying finite field [9].

The set of points $\{(x, y) \in E(\mathbb{F}_{2^m})\} \cup \{\mathcal{O}\}$ under the addition operation $+$ (chord and tangent) forms an additive group, with $\mathcal{O}$ as the identity element. Given an elliptic point $P \in E(\mathbb{F}_{2^m})$ and an integer k, the operation kP, called *point multiplication*, is defined by the addition of the point P to itself $k - 1$ times:

$$kP = \underbrace{P + P + \ldots + P}_{k-1 \text{ additions}}.$$

Public key cryptography protocols, such as the Elliptic Curve Diffie-Hellman key agreement [3] and the Elliptic Curve Digital Signature Algorithm [3], employ point multiplication as a fundamental operation; and their security is based on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP). This problem consists in finding the discrete logarithm k given a point kP. Criteria for selecting suitable secure curves are a complex subject and a matter of much discussion. We adopt the well-known standard NIST curves as a conservative choice, but we refer the reader to [3] for further details on how to generate efficient curves where instances of the ECDLP are computationally hard.

We restrict the discussion to non-supersingular curves because supersingular curves are not suitable for elliptic curve cryptosystems based on the ECDLP problem [21]. However, supersingular curves are particularly of interest in applications of pairing-based protocols on WSNs [25].

4. The platform

The MICAz Mote sensor node is equipped with an ATmega128 8-bit processor clocked at 7.3728MHz. The program code is loaded from an 128KB EEPROM chip

and runtime memory is stored in a 4KB RAM chip [10]. The ATmega128 processor is a typical RISC architecture with 32 registers, but six of them are special pointer registers. Since at least one register is needed to store temporary results or data loaded from memory, 25 registers are generally available for arithmetic. The instruction set is also reduced, as only 1-bit shift/rotate instructions are natively supported. Bitwise shifts by arbitrary amounts can then be implemented with combinations of shift/rotate instructions and other instructions. The processor pipeline has two stages and memory instructions always cause pipeline stalls. Arithmetic instructions with register operands cost 1 cycle and memory instructions or memory addressing cost 2 processing cycles [1]. Table 2 presents the instructions provided by the platform which can be used for the implementation of binary field arithmetic.

Instruction	Description	Use	Cost
`rsl`, `lsl`	Right/left 1-bit shift	Multi-precision 1-bit shift	1 cycle
`rol`, `ror`	Right/left 1-bit rotate	Multi-precision 1-bit shift	1 cycle
`swap`	Swap high and low nibbles	Shift by 4 bits	1 cycle
`bld`, `bst`	Bit load/store from/to flag	Shift by 7 bits	1 cycle
`eor`	Bitwise exclusive `OR`	Binary field addition	1 cycle
`ld`, `st`	Memory load/store	Read operands/write results	2 cycles
`adiw`, `sbiw`	Pointer arithmetic	Memory addressing	2 cycles

Table 2. Relevant instructions for the implementation of binary field arithmetic.

5. Algorithms for finite field arithmetic

In this section we will represent the elements of $\mathbb{F}_{2^m}$ using a polynomial basis. Let $f(z)$ be an irreducible binary trinomial or pentanomial of degree m. The elements of $\mathbb{F}_{2^m}$ are the binary polynomials of degree at most $m-1$. A field element $a(z) = \sum_{i=0}^{m-1} a_i z^i$ is associated with the binary vector $a = (a_{m-1}, \ldots, a_1, a_0)$ of length m. In a software implementation in an 8-bit processor, the element a is stored as a vector of $n = \lceil m/8 \rceil$ bytes. The field operations in $\mathbb{F}_{2^m}$ can be implemented by common processor instructions, such as logical shifts ($\gg$,$\ll$) and addition modulo 2 (XOR, $\oplus$).

5.1. Multiplication. The computation of kP is the most time-consuming operation on ECC and this operation depends directly on the finite field arithmetic. In particular, a fast field multiplication is critical for the performance of ECC.

Two different strategies are commonly considered for the implementation of multiplication in $\mathbb{F}_{2^m}$. The first one consists in applying the Karatsuba's algorithm [11] to divide the multiplication in sub-problems and solve each problem independently by the following formula [9] (with $a(z) = A_1 z^{\lceil m/2 \rceil} + A_0$ and $b(z) = B_1 z^{\lceil m/2 \rceil} + B_0$):

$$c(z) = a(z) \cdot b(z) = A_1 B_1 z^m + [(A_1 + A_0)(B_1 + B_0) + A_1 B_1 + A_0 B_0] z^{\lceil m/2 \rceil} + A_0 B_0.$$

Naturally, Karatsuba multiplication imposes some overhead for the divide and conquer steps. The second one consists in applying a direct algorithm like the López-Dahab (LD) binary field multiplication (Algorithm 1) [19]. In this algorithm, the precomputation window is usually chosen as $t = 4$ and the precomputation table T has size $|T| = 16(n+1)$, since each element $T[i]$ requires at most $n+1$ bytes to store the result of $u(z)b(z)$. Operand a is scanned from left to right and processed in groups of 4 bits. In an 8-bit processor, the algorithm is comprised by two

phases, where the lower halves of bytes of a are processed in the first phase and the higher halves are processed in the second phase. These phases are separated by an intermediate shift which implements multiplication by z^t.

Algorithm 1 López-Dahab multiplication in $\mathbb{F}_{2^m}$ [19].

Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..2n-1]$.

1: Compute $T(u) = u(z)b(z)$ for all polynomials $u(z)$ of degree lower than t.
2: $c[0 \ldots 2n-1] \leftarrow 0$
3: **for** $k \leftarrow 0$ to $n-1$ **do**
4: $\quad u \leftarrow a[k] \gg t$
5: $\quad$ **for** $j \leftarrow 0$ to n **do**
6: $\quad\quad c[j+k] \leftarrow c[j+k] \oplus T(u)[j]$
7: $\quad$ **end for**
8: **end for**
9: $c(z) \leftarrow c(z)z^t$
10: **for** $k \leftarrow 0$ to $n-1$ **do**
11: $\quad u \leftarrow a[k] \mod 2^t$
12: $\quad$ **for** $j \leftarrow 0$ to n **do**
13: $\quad\quad c[j+k] \leftarrow c[j+k] \oplus T(u)[j]$
14: $\quad$ **end for**
15: **end for**
16: **return** c

Conventionally, the series of additions involved in the LD multiplication are implemented through additions over subparts of a double-precision vector. In order to reduce the number of memory accesses employed during these additions, we employ a rotating register window. This window simulates the series of additions by accumulating consecutive writes into registers. After a final result is obtained in the lowest precision register, this value is written into memory and this register is free to participate as the highest precision register. Figure 1 shows a rotating register window with $n+1$ registers. We modify the LD multiplication algorithm by integrating a rotating register window. The result of this integration is referred as *LD multiplication with registers* and shown as Algorithm 2. Figure 2 presents this modification graphically. These descriptions of the algorithm assumes that n general-purpose registers are available for arithmetic. If this is not the case, (e.g. multiplication in $\mathbb{F}_{2^{233}}$ on this platform) the accumulation in the register window must be divided in different blocks in a multistep fashion and each block processed with a different rotating register window. A slight overhead is introduced between the processing of consecutive blocks because some registers must be written into memory and freed before they can be used in a new rotating register window.

An additional suggested optimization is the separation of the precomputation table T in different blocks of 256 bytes, where each block is stored on a 256-byte aligned memory address. This optimization accelerates memory addressing because offsets lower than 256 can be computed by a simple 1-cycle addition instruction, avoiding expensive pointer arithmetic. Another optimization is to store the results of the first phase of the algorithm already shifted, eliminating some redundant memory reads to reload the intermediate result into registers for multi-precision shifting. A last optimization is the embedding of modular reduction at the end of

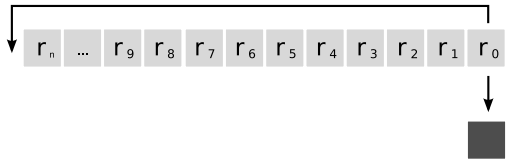

FIGURE 1. Rotating register window with $n+1$ registers.

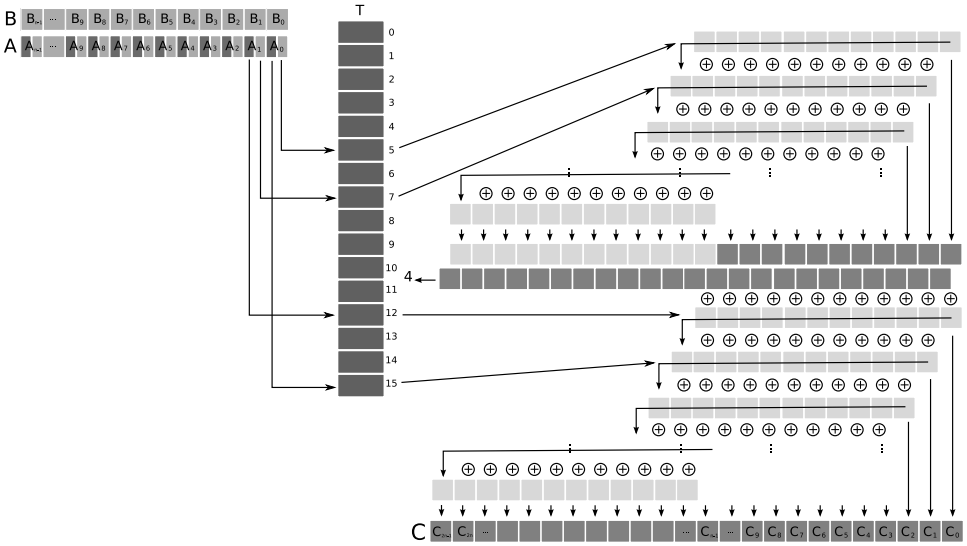

FIGURE 2. López-Dahab multiplication with registers of two field elements represented as n-byte vectors in an 8-bit processor.

Algorithm 2 Proposed optimization for multiplication in $\mathbb{F}_{2^m}$ using $n+1$ registers.

Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..2n-1]$.
Note: v_i denotes the vector of $n+1$ registers $(r_{i-1}, \ldots, r_0, r_n, \ldots, r_i)$.

1: Compute $T(u) = u(z)b(z)$ for all polynomials $u(z)$ of degree lower than 4.
2: Let u_i be the 4 most significant bits of $a[i]$.
3: $v_0 \leftarrow T(u_0)$, $c[0] \leftarrow r_0$
4: $v_1 \leftarrow v_1 \oplus T(u_1)$, $c[1] \leftarrow r_1$
5: $\cdots$
6: $v_{n-1} \leftarrow v_{n-1} \oplus T(u_{n-1})$, $c[n-1] \leftarrow r_{n-1}$
7: $c \leftarrow ((r_{n-2}, \ldots, r_0, r_n) \mid\mid (c[n-1], \ldots, c[0])) \ll 4$
8: Let u_i be the 4 least significant bits of $a[i]$.
9: $v_0 \leftarrow T(u_0)$, $c[0] \leftarrow c[0] \oplus r_0$
10: $\cdots$
11: $v_{n-1} \leftarrow v_{n-1} \oplus T(u_{n-1})$, $c[n-1] \leftarrow c[n-1] \oplus r_{n-1}$
12: $c[n \ldots 2n-1] \leftarrow c[n \ldots 2n-1] \oplus (r_{n-2}, \ldots, r_0, r_n)$
13: **return** c

the multiplication algorithm. This trick allows the reuse of values already loaded into registers to speed up modular reduction. The following analysis does not take these suggested optimizations into account.

ANALYSIS OF MULTIPLICATION ALGORITHMS. Observing the fact that the more expensive instructions in the target platform are related to memory accesses, the behavior of different algorithms was analyzed to estimate their performance. This analysis traces the cost of different algorithms in terms of memory accesses (reads and writes) and arithmetic instructions (XOR).

Without considering partial multiplications, the Karatsuba algorithm in a binary field executes approximately $11n$ memory reads, $7n$ memory writes and $4n$ XOR instructions.

For LD multiplication, analysis shows that building the precomputation table requires n memory reads to obtain the values $b[i]$ and $|T|$ writes and $11n$ XOR instructions for filling the table. Inside each inner loop, the algorithm executes $2(n+1)$ memory reads, $n+1$ writes and $n+1$ XOR instructions. In each outer loop, the algorithm executes n memory accesses to read the values $a[k]$ and n iterations of the inner loop, totalizing $n+2n(n+1)$ reads, $n(n+1)$ writes and $n(n+1)$ XOR instructions. The logical shift of $c(z)$ computed at the intermediate stage requires $2n$ memory reads and writes. Considering the initialization of c, we have $3n+2(n+2n(n+1))$ memory reads, $|T|+2(2n)+2n(n+1)$ writes and $11n+2n(n+1)$ XOR instructions.

For the proposed optimization (Algorithm 2), building the precomputation table requires n memory reads to obtain the values $b[i]$ and $|T|$ writes and $11n$ XOR instructions for filling the table. Line 3 of the algorithm executes $n+1$ memory reads and 1 write on $c[0]$. Lines 4-6 execute $n+1$ memory reads, 1 write on $c[i]$ and $n+1$ XOR instructions, all this $n-1$ times. The intermediate shift executes n reads and $(2n)$ writes. Lines 9-11 execute $n+1$ memory reads, 1 read and write on $c[i]$ and $n+2$ XOR instructions, all this n times. The final operation costs n memory reads, writes and XOR instructions. The algorithm thus requires a total of $3n+n(n+1)+n(n+2)$ reads, $|T|+n+2n+2n$ writes and $11n+(n-1)(n+1)+n(n+2)+n$ XOR instructions.

Table 3 presents the costs associated with memory operations for LD multiplication, LD with registers multiplication and Karatsuba multiplication. Table 4 presents approximate costs of the algorithms in terms of executed memory instructions for the fields $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$.

	Number of instructions in terms of vectors of n bytes		
Method	Reads	Writes	XOR
López-Dahab	$4n^2+9n$	$\|T\|+2n^2+6n$	$2n^2+13n$
LD with registers	$2n^2+6n$	$\|T\|+5n$	$2n^2+14n-1$
Karatsuba	$11n+3M(\lceil n/2\rceil)$	$7n+3M(\lceil n/2\rceil)$	$4n+3M(\lceil n/2\rceil)$

TABLE 3. Costs in number of executed instructions for the multiplication algorithms in $\mathbb{F}_{2^m}$. $M(x)$ denotes the cost of a multiplication algorithm which multiplies two x-byte vectors.

We can see from Table 3 that the number of memory accesses for LD with registers is drastically reduced in comparison with the original algorithm, reducing the number of reads by half and the number of writes by a quadratic factor. The comparison between LD with registers and Karatsuba+LD with registers favors the first (lower number of writes) on both finite fields. One problem with this

	$n = 21$			$n = 30$		
Method	Reads	Writes	XOR	Reads	Writes	XOR
López-Dahab	1953	1452	1155	3870	2476	2190
LD with registers	1071	457	1175	1980	646	2219
Karatsuba+LD	1980	1647	1239	3310	2518	1984
Karatsuba+LD with registers	1155	888	1269	1898	1134	2025

TABLE 4. Costs in number of executed instructions for the multiplication algorithms in $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$. The Karatsuba algorithm in $\mathbb{F}_{2^{233}}$ executes two instances of cost $M(15)$ and one instance of cost $M(14)$ to better approximate the results.

analysis is that it assumes that the processor has at least n general-purpose registers available for arithmetic. This is not true in $\mathbb{F}_{2^{233}}$, because the algorithm requires 31 registers for a full rotating register window. The decision between a multistep implementation of LD with registers and Karatsuba+LD with registers will depend on the actual implementation of the algorithms.

5.2. MODULAR REDUCTION. The NIST irreducible polynomial for the finite field $\mathbb{F}_{2^{163}}$, $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$, allows a fast modular reduction algorithm. Algorithm 3 [29] presents an adaptation of this algorithm for 8-bit processors. In this algorithm, reducing a digit $c[i]$ of the upper half of the vector c requires six memory accesses to read and write $c[i]$ on lines 3-5. Four of them are redundant because ideally we only need to read and write $c[i]$ once. We eliminate these redundant accesses by employing a rotating register window of three registers which accumulate writes into registers before a final result can be written into memory. This optimization is given in Algorithm 4 along with the substitution of some bitwise shifts which are expensive in this platform for cheaper ones. Since the processor only supports 1-bit and 4-bit shifts natively, we further replace the various expensive shifts in the accumulate function R by table lookups on 256-byte tables. These tables are stored on 256-byte aligned memory addresses to speed up memory addressing. The new version of the accumulate function is depicted in Algorithm 5.

Algorithm 3 Fast modular reduction by $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$.

Input: $c(z) = c[0..40]$.
Output: $c(z) \bmod f(z) = c[0..20]$.

1: **for** $i \leftarrow 40$ **downto** 21 **do**
2: $\quad t \leftarrow c[i]$
3: $\quad c[i-19] \leftarrow c[i-19] \oplus (t \gg 4) \oplus (t \gg 5)$
4: $\quad c[i-20] \leftarrow c[i-20] \oplus (t \ll 4) \oplus (t \ll 3) \oplus t \oplus (t \gg 3)$
5: $\quad c[i-21] \leftarrow c[i-21] \oplus (t \ll 5)$
6: **end for**
7: $t \leftarrow c[20] \gg 3$
8: $c[0] \leftarrow c[0] \oplus (t \ll 7) \oplus (t \ll 6) \oplus (t \ll 3) \oplus t$
9: $c[1] \leftarrow c[1] \oplus (t \gg 1) \oplus (t \gg 2)$
10: $c[20] \leftarrow c[20] \wedge \texttt{0x07}$
11: **return** c

For the NIST irreducible polynomial in $\mathbb{F}_{2^{233}}$ on 8-bit processors, we present Algorithm 6, a direct adaptation of the standard algorithm. This algorithm only

Algorithm 4 Fast modular reduction in $\mathbb{F}_{2^{163}}$ with rotating register window.

Input: $c(z) = c[0..40]$.
Output: $c(z) \bmod f(z) = c[0..20]$.
Note: The accumulate function $R(r_0, r_1, r_2, t)$ executes:

$$s_0 \leftarrow t \ll 4$$
$$r_0 \leftarrow (r_0 \oplus t \oplus (t \gg 1)) \gg 4$$
$$r_1 \leftarrow r_1 \oplus s_0 \oplus (t \ll 3) \oplus t \oplus (t \gg 3)$$
$$r_2 \leftarrow s_0 \ll 1$$

1: $r_b \leftarrow 0$, $r_c \leftarrow 0$
2: **for** $i \leftarrow 40$ **downto** 25 **by** 3 **do**
3: $\quad R(r_b, r_c, r_a, c[i])$, $c[i-19] \leftarrow c[i-19] \oplus r_b$
4: $\quad R(r_c, r_a, r_b, c[i-1])$, $c[i-20] \leftarrow c[i-20] \oplus r_c$
5: $\quad R(r_a, r_b, r_c, c[i-2])$, $c[i-21] \leftarrow c[i-21] \oplus r_a$
6: **end for**
7: $R(r_b, r_c, r_a, c[22])$, $c[3] \leftarrow c[3] \oplus r_b$
8: $R(r_c, r_a, r_b, c[21])$, $c[2] \leftarrow c[2] \oplus r_c$
9: $r_a \leftarrow c[1] \oplus r_a$
10: $r_b \leftarrow c[0] \oplus r_b$
11: $t \leftarrow c[20]$
12: $c[20] \leftarrow t \wedge \texttt{0x07}$
13: $t \leftarrow t \gg 3$
14: $c[0] \leftarrow r_b \oplus (t \ll 7) \oplus (t \ll 6) \oplus (t \ll 3) \oplus t$
15: $c[1] \leftarrow r_a \oplus (t \gg 1) \oplus (t \gg 2)$
16: **return** c

Algorithm 5 Optimized version of the accumulate function R.

Input: r_0, r_1, r_2, t.
Output: r_0, r_1, r_2.

1: $r_0 \leftarrow r_0 \oplus T_0[t]$
2: $r_1 \leftarrow r_1 \oplus T_1[t]$
3: $r_2 \leftarrow t \ll 5$

executes 1-bit or 7-bit shifts. These two shifts can be translated efficiently to the processor instruction set, because 1-bit shifts are supported natively and 7-bit shifts can be emulated efficiently. Hence lookup tables are not needed and the only optimization made during implementation of Algorithm 6 was complete unrolling of the main loop and straightforward elimination of consecutive redundant memory accesses.

ANALYSIS OF MODULAR REDUCTION ALGORITHMS. As pointed by Seo et al. [29], Algorithm 3 executes many redundant memory accesses: 4 memory reads and 3 writes during each loop iteration and additional 4 reads and 3 writes on the final step, which sum up to 88 reads and 66 writes. The proposed optimization reduces the number of memory operations to 43 reads and 23 writes. Despite Algorithm 5 being specialized for the chosen polynomial, the register window technique can be applied to any irreducible polynomial with the non-null coefficients located in the first word. The implementation of Algorithm 6 also reduces the number of memory

Algorithm 6 Fast modular reduction by $f(z) = z^{233} + z^{74} + 1$.

Input: $c(z) = c[0..58]$.
Output: $c(z) \bmod f(z) = c[0..29]$.

1: **for** $i \leftarrow 58$ **downto** 32 **by** 2 **do**
2: $t_0 \leftarrow c[i]$
3: $t_1 \leftarrow c[i-1]$
4: $c[i-19] \leftarrow c[i-19] \oplus (t_0 \gg 7)$
5: $c[i-20] \leftarrow c[i-20] \oplus (t_0 \ll 1) \oplus (t_1 \gg 7)$
6: $c[i-21] \leftarrow c[i-21] \oplus (t_1 \ll 1)$
7: $c[i-29] \leftarrow c[i-29] \oplus (t_0 \gg 1)$
8: $c[i-30] \leftarrow c[i-30] \oplus (t_0 \ll 7) \oplus (t_1 \gg 1)$
9: $c[i-31] \leftarrow c[i-31] \oplus (t_1 \ll 7)$
10: **end for**
11: $t_0 \leftarrow c[30]$
12: $c[0] \leftarrow c[0] \oplus (t_0 \ll 7)$
13: $c[1] \leftarrow c[1] \oplus (t_0 \gg 1)$
14: $c[10] \leftarrow c[10] \oplus (t_0 \ll 1)$
15: $c[11] \leftarrow c[11] \oplus (t_0 \gg 7)$
16: $t_0 \leftarrow c[29] \gg 1$
17: $c[0] \leftarrow c[0] \oplus t_0$
18: $c[9] \leftarrow c[9] \oplus (t_0 \ll 2)$
19: $c[10] \leftarrow c[10] \oplus (t_0 \gg 6)$
20: $c[29] \leftarrow c[29] \wedge \texttt{0x01}$
21: **return** c

accesses, since a standard implementation executes 122 reads and 92 writes while our implementation executes 92 memory reads and 62 writes.

5.3. SQUARING. The square of a finite field element $a(z) \in \mathbb{F}_{2^m}$ is given by $a(z)^2 = \sum_{i=0}^{m-1} a_i z^{2i} = a_{m-1} z^{2m-2} + \cdots + a_2 z^4 + a_1 z^2 + a_0$. The binary representation of $a(z)^2$ can be computed by inserting a "0" bit between each pair of successive bits on the binary representation of $a(z)$ and accelerated by introducing a 16-byte lookup table. If modular reduction is computed in a separate step, reduntant memory operations are required to store the squaring result and reload this result for reduction. This can be improved by embedding the modular reduction step directly into the squaring algorithm. This way, the lower half of the digit vector a is expanded in the usual fashion and the upper half digits are expanded and immediately reduced. If modular reduction of a single byte requires expensive shifts, additional lookup tables can be used to store the expanded bytes already reduced. This is illustrated in Algorithm 7 which computes squaring in $\mathbb{F}_{2^{163}}$ using the same small rotating register window as Algorithm 5 and three additional 16-byte lookup tables T_0, T_1 and T_2. For squaring in $\mathbb{F}_{2^{233}}$, we also combine byte expansion of the digit vector's lower half with Algorithm 6 for fast reduction.

5.4. INVERSION. For inversion in $\mathbb{F}_{2^m}$ we implemented the Extended Euclidean Algorithm for polynomials [9]. Since this algorithm requires flexible left shifts by arbitrary amounts, we implemented six dedicate shifting functions to shift a binary field element by every amount possible for an 8-bit processor. The core of a multi-precision left shift algorithm is the sequence of instructions which receives as input

Algorithm 7 Squaring in $\mathbb{F}_{2^{163}}$.

Input: $a(z) = a[0..20]$.
Output: $c(z) = a(z)^2 \bmod f(z)$.
Note: The accumulate function $R(r_0, r_1, r_2, t)$ executes:

$$r_0 \leftarrow r_0 \oplus T_0[t], r_1 \leftarrow r_1 \oplus T_1[t], r_2 \leftarrow r_2 \oplus T_2[t]$$

1: For each 4-bit combination u, $T(u) = (0, u_3, 0, u_2, 0, u_1, 0, u_0)$.
2: **for** $i \leftarrow 0$ to 9 **do**
3: $\quad c[2i] \leftarrow T(a[i] \wedge \texttt{0x0F})$
4: $\quad c[2i+1] \leftarrow T(a[i] \gg 4)$
5: **end for**
6: $c[20] \leftarrow T(a[10] \wedge \texttt{0x0F})$
7: $r_b \leftarrow 0$, $r_c \leftarrow 0$, $j \leftarrow 20$
8: $t_0 \leftarrow a[20] \wedge \texttt{0x0F}$
9: $R(r_b, r_c, r_a, t_0)$, $c[21] \leftarrow r_b$
10: **for** $i \leftarrow 19$ **downto** 13 **by** 3 **do**
11: $\quad a_o \leftarrow a[i]$, $t_0 \leftarrow a_0 \gg 4$, $t_1 \leftarrow a_0 \wedge \texttt{0x0F}$
12: $\quad R(r_c, r_a, r_b, t_0)$, $c[j] \leftarrow c[j] \oplus r_c$
13: $\quad R(r_a, r_b, r_c, t_1)$, $c[j-1] \leftarrow c[j-1] \oplus r_a$
14: $\quad a_0 \leftarrow a[i-1]$, $t_0 \leftarrow a_0 \gg 4$, $t_1 = a_0 \wedge \texttt{0x0F}$
15: $\quad R(r_b, r_c, r_a, t_0)$, $c[j-2] \leftarrow c[j-2] \oplus r_b$
16: $\quad R(r_c, r_a, r_b, t_1)$, $c[j-3] \leftarrow c[j-3] \oplus r_c$
17: $\quad a_0 = a[i-2]$, $t_0 = a_0 \gg 4$, $t_1 = a_0 \wedge \texttt{0x0F}$
18: $\quad R(r_a, r_b, r_c, t_0)$, $c[j-4] \leftarrow c[j-4] \oplus r_a$
19: $\quad R(r_b, r_c, r_a, t_1)$, $c[j-5] \leftarrow c[j-5] \oplus r_b$
20: $\quad j \leftarrow j - 6$
21: **end for**
22: $t_0 = a[10] \gg 4$
23: $R(r_c, r_a, r_b, t_0), c[2] \leftarrow c[2] \oplus r_c$
24: $r_a \leftarrow c[1] \oplus r_a, r_b \leftarrow c[0] \oplus r_b$
25: $t \leftarrow c[21]$
26: $r_a \leftarrow r_a \oplus t \oplus (t \ll 3) \oplus (t \ll 4) \oplus (t \gg 3)$
27: $r_b \leftarrow r_b \oplus (t \ll 5)$
28: $t \leftarrow c[20]$
29: $c[20] \leftarrow t \wedge \texttt{0x07}$
30: $t \leftarrow t \gg 3$
31: $c[0] \leftarrow r_b \oplus (t \ll 7) \oplus (t \ll 6) \oplus (t \ll 3) \oplus t$
32: $c[1] \leftarrow r_a \oplus (t \gg 1) \oplus (t \gg 2)$
33: **return** c

the amount to shift i, a register r and a carry register rc storing the bits shifted out in the last iteration; and produce $(r \ll i) \oplus rc$ as output and $r \gg (8-i)$ as new carry. Table 5 lists the required instructions and costs in cycles for shifting a single byte in each of the implemented multi-precision shifts by i bits. Each instruction in the table cost 1 cycle, thus the cost to compute the core of a multi-precision left shift by i bits is just the number of rows in the i-th row of the table.

6. Algorithms for elliptic curve arithmetic

We have selected fast algorithms for elliptic curve arithmetic in three situations: multiplying a random point P by a scalar k, multiplying the generator G by a scalar k and simultaneously multiplying two points P and Q by scalars k and l to obtain

i	Intructions
1	`rol r`

2	`clr rt`
	`lsl r`
	`rol rt`
	`lsl r`
	`rol rt`
	`eor r, rc`
	`mov rc, rt`

i	Intructions

3	`clr rt`
	`lsl r`
	`rol rt`
	`lsl r`
	`rol rt`
	`lsl r`
	`rol rt`
	`eor r, rc`
	`mov rc, rt`

i	Intructions

4	`swap r`
	`mov rt, r`
	`andi r, 0xF0`
	`andi rt, 0x0F`
	`eor r, rc`
	`mov rc, rt`

5	`swap r`
	`mov rt, r`
	`andi r, 0xF0`
	`andi rt, 0x0F`
	`lsl r`
	`rol rt`
	`eor r, rc`
	`mov rc, rt`

i	Intructions

6	`bst rt, 0`
	`bld r, 6`
	`bst rt, 1`
	`bld r, 7`
	`lsr rt`
	`lsr rt`
	`eor r, rc`
	`mov rc, rt`

7	`bst rt, 0`
	`bld r, 7`
	`lsr rt`
	`eor r, rc`
	`mov rc, rt`

TABLE 5. Processor instructions used to efficiently implement multi-precision left shifts by i bits. The input register is `r`, the carry register is `rc` and a temporary register is `rt`. When $i = 1$, `rc` is represented by the carry processor flag.

$kP + lQ$. Our implementation uses mixed addition with projective coordinates [18], given that the ratio of inversion to multiplication is 16.

For multiplying a random point by a scalar, we choose Solinas' *τ-adic non-adjacent form* (TNAF) representation [30] with $w = 4$ for Koblitz curves (4-TNAF method with 4 precomputation points) and the method due to López and Dahab [17] for random binary curves. Solinas' algorithm explores the optimizations provided by Koblitz curves and accelerates the computation of kP by substituting point doublings for applications of the efficiently computable endomorphism based on the Frobenius map $\tau(x, y) = (x^2, y^2)$. The method due to López and Dahab does not use precomputation, its execution time is constant and each iteration of the algorithm executes the same number of operations, independently of the bit pattern in k [9].

For multiplying the generator, we employ the same 4-TNAF method for Koblitz curves; and for generic curves, we employ the Comb method [16] with 16 precomputed points. Precomputed tables for the generator are stored in ROM memory to reduce RAM consumption. Larger precomputed tables can be used if program size is not an issue.

For simultaneous multiplication, we implement the interleaving method with 4-TNAFs for Koblitz curves and the interleaving of 4-NAFs with integers represented in *non-adjacent form* (NAF) for generic curves [6]. The same table built for multiplying the generator is used during simultaneous multiplication in Koblitz curves when point P or Q is the generator G. An additional small table of 4 points is precomputed for the generator and stored in ROM to provide the same situation with generic curves.

7. IMPLEMENTATION RESULTS

The compiler and assembler used is the GCC 4.1.2 suite for ATmega128 with optimization level `-O2`. The timings were measured with the software AVR Studio 4.14 [2]. This tool is a cycle-accurate simulator frequently used to prototype software for execution on the target platform. We have written a specialized library containing the software implementations.

FINITE FIELD ARITHMETIC. The algorithms for squaring, multiplication, modular reduction and inversion in the finite field were implemented in the C language and Assembly. Table 6 presents the costs measured in cycles of each implemented operation in $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$. Since the platform does not have cache memory or out-of-order execution, the finite field operations always cost the same number of cycles and the timings were taken exactly once, except for inversion. The timing for inversion was taken as the average of 50 timings measured on consecutive executions of the algorithm.

	$m = 163$		$m = 233$	
Algorithm	C language	*Assembly*	C language	*Assembly*
Squaring	629	430	908	463
Modular Squaring	1154	570	1340	956
LD Mult. with registers	13838	4508	–	8314
LD Mult. (new variant)	9738	–	18028	–
Karatsuba+LD with registers	12246	6968	25850	9261
Modular reduction	606	430	911	620
Inversion	243790	81365	473618	142986

TABLE 6. Timings in cycles for arithmetic algorithms in $\mathbb{F}_{2^m}$.

From Table 6, $m = 163$, we can observe that in the C language implementation, Karatsuba+LD with registers multiplication is more efficient than the direct application of LD with registers multiplication. This contradicts the preliminary analysis based on the number of memory accesses executed by each algorithm. This can be explained by the fact that the LD with registers multiplication uses 21 of the 32 general-purpose registers to store intermediate results during multiplication. Several additional registers are also needed to store memory addresses and temporary variables for arithmetic operations. The inefficiency found is thus originated from the difficulty of the C compiler to maintain all intermediate values on registers. To confirm this limitation, a new variant of LD with registers multiplication which reduces the number of temporary variables needed was also implemented. This variant processes 32 bits of the operand in each interaction compared to the original version of LD multiplication which processes 4 bits in each interaction. The new variant reduces the number of memory accesses while keeping a smaller number of temporary variables and thus exhibits the expected performance. For the squaring algorithm, we can see that embedding the modular reduction step reduces the cost of modular squaring significantly compared with the sequential execution of squaring plus modular reduction. Table 6, $m = 233$, shows that the Karatsuba algorithm in $\mathbb{F}_{2^{233}}$ indeed does not improve performance over the multistep implementation of LD with registers multiplication, even if the processor does not have enough registers to store the full rotating register window. The Assembly implementations demonstrate the compiler inefficiency in generating optimized code and allocating resources for the target platform, showing considerably faster timings.

ELLIPTIC CURVE ARITHMETIC. Point multiplication was implemented on elliptic curves standardized by NIST. Table 7 presents the execution time of the multiplication of a random point P by a random integer k of 163 or 233 bits, with the underlying finite field arithmetic implemented in C or Assembly. In each of the programming languages, the fastest field multiplication algorithm is used. The results

were computed by the arithmetic mean of the timings measured on 50 consecutive executions of the algorithm.

	C language			*Assembly*		
Curve	kG	kP	$kP + lQ$	kG	kP	$kP + lQ$
NIST-K163 (Koblitz)	0.56	0.67	1.24	0.29	0.32	0.60
NIST-B163 (Generic)	0.77	1.55	2.21	0.37	0.74	1.04
NIST-K233 (Koblitz)	1.26	1.48	2.81	0.66	0.73	1.35
NIST-B233 (Generic)	1.94	3.90	5.35	0.94	1.89	2.52

Table 7. Timings in seconds for point multiplication.

Table 8 compares the performance of the proposed implementation with TinyECCK [29] and the work of Kargl et al. [12], the previously fastest binary curves implementation in C and Assembly published for this platform. For the C implementation, we achieve faster timings on all finite field arithmetic operations with improvements over 50%. For the Assembly implementation, we obtain speed improvements on field squaring and multiplication and exactly the same timing for modular reduction, but the polynomial used by Kargl et al.[12] is a trinomial carefully selected to support a faster modular reduction algorithm. The computation of kP on Koblitz curves implemented in C language was 41% faster than TinyECCK. By choosing the López-Dahab point multiplication algorithm with generic curves implemented in Assembly, we achieve a timing 11% faster than [12] while satisfying the timing-resistant property. If we relax this condition, we obtain a point multiplication 61% faster in Assembly by using Solinas' method. Comparing our Assembly implementation with TinyECCK and [12] with the same curve parameters, we achieve a 72% speedup and an 11% speedup for point multiplication, respectively.

	Proposed	**TinyECCK**	**Proposed**	**Kargl et al.** [12]
Algorithm	C language	C language	Assembly	Assembly
Modular Squaring	1154 c	2729 c	570 c	663
Multiplication	9738 c	19670 c	4508 c	5057 c
Modular reduction	606 c	1904 c	430 c	433 c
Inversion	243790 c	539132 c	81365 c	–
kP on Koblitz	0.67 s	1.14 s	0.32 s	–
kP on Generic	1.55 s	–	0.74 s	0.83 s

Table 8. Comparison between different implementations. The timings are presented in cycles (c) or seconds (s) on a 7.2838MHz device.

The fastest time for point multiplication previously published for this platform at the 160-bit security level was 0.745 second [7]. Compared to this implementation, which uses prime fields, the proposed optimizations result in a point multiplication 57% faster.

The implemented optimizations allow performance gains but provoke a collateral effect on memory consumption. Table 9 presents memory requirements for code size and RAM memory for the different implementations at the 160-bit security level. We can also observe that Assembly implementations are responsible for a significant expansion in program code size.

	ROM memory	Static RAM	Stack RAM
Proposed (Koblitz) – C	22092	1028	1207
Proposed (Koblitz) – C+Assembly	25802	1732	1207
Proposed (Generic) – C	12848	881	682
Proposed (Generic) – C+Assembly	16218	1585	682
TinyECCK (C-only)	5592	–	618
Kargl et a. (C+Assembly) [12]	11264	–	–

TABLE 9. Cost in *bytes* of memory for implementations of scalar multiplication of a random point at the 160-bit security level.

CRYPTOGRAPHIC PROTOCOLS. We now illustrate the performance obtained by our efficient implementation with some executions of cryptographic protocols for key agreement and digital signatures. Key agreement is employed in sensor networks for establishing symmetric keys which can be used for encryption or authentication. Digital signatures are employed for communication between the sensor nodes and the base stations where data must be made available to multiple applications and users [24]. For key agreement between nodes, we implemented the Elliptic Curve Diffie & Hellman (ECDH) protocol [3], and for digital signatures, we implemented the Elliptic Curve Digital Signature Algorithm (ECDSA) [3]. We assume that public and private keys are generated and loaded into the nodes before the deployment of the sensor network. Hence timings for key generation and public key authentication are not presented or considered. Table 10 presents the timings for the ECDH protocol and Table 11 presents the timings for the ECDSA protocol, using the choice of algorithms discussed in Section 6. Results on these tables pose an interesting decision between deploying generic binary curves on the lower security level or deploying special curves on the higher security level.

	C language			*Assembly*		
Curve	Time	ROM	RAM	Time	ROM	RAM
NIST-K163	0.74	28.3	2.2	0.39	32.0	2.8
NIST-B163	1.62	24.0	1.1	0.81	27.8	1.9
NIST-K233	1.55	31.0	2.9	0.80	38.6	3.7
NIST-B233	3.97	26.9	1.5	1.96	34.6	2.2

TABLE 10. Timings for the ECDH protocol execution. Timings are given in seconds and ROM memory or Static+Stack RAM consumption are given in KB.

	C language			*Assembly*		
Curve	Time (S + V)	ROM	RAM	Time (S + V)	ROM	RAM
NIST-K163	0.67 + 1.23	31.8	2.9	0.36 + 0.63	35.3	3.7
NIST-B163	0.87 + 2.17	29.6	2.1	0.45 + 1.05	33.2	2.8
NIST-K233	1.46 + 2.76	34.6	3.1	0.78 + 1.39	42.2	3.8
NIST-B233	2.09 + 5.25	32.8	2.3	1.04 + 2.55	40.4	3.1

TABLE 11. Timings for the ECDSA protocol execution. Timings for signature (S) and verification (V) are given in seconds and ROM memory or Static+Stack RAM consumption are given in KB.

8. Conclusions

Despite several years of intense research, security and cryptography on WSNs still face several open problems. In this work, we presented efficient implementations of binary field algorithms such as squaring, multiplication, modular reduction and inversion. These implementations take into account the characteristics of the target platform (the MICAz Mote) to develop optimizations, specifically: (i) the cost of memory addressing; (ii) the cost of memory instructions; (iii) the limited flexibility of bitwise shift instructions. We obtain the fastest binary field arithmetic implementations in C and Assembly published for the target platform. Significant performance benefits where achieved by the Assembly implementation, resulting from fine-grained resource allocation and instruction selection. These optimizations produced a point multiplication at the 160-bit security level under $\frac{1}{3}$ of a second, an improvement of 72% compared to the best implementation of a Koblitz curve previously published and an improvement of 61% compared to the best implementation of binary curves. When compared to the best implementation of prime curves, we obtain a performance gain of 57%. We also presented the first timings of elliptic curves at the higher 233-bit security level. For both security levels, we illustrate the performance obtained with executions of key agreement and digital signature protocols. In particular, a key agreement can be computed in under 0.40 second at the 163-bit security level and under 0.80 second at the 233-bit security level. A digital signature can be computed and verified in 1 second at the 163-bit security level and in 2.17 seconds at the 233-bit security level. We hope that our results can increase the efficiency and viability of elliptic curve cryptography on wireless sensor networks.

Acknowledgements

We would like to thank the referees for their valuable comments and suggestions. Diego F. Aranha is supported by FAPESP, grant no. 2007/06950-0. Julio López and Ricardo Dahab are partially supported by CNPq and FAPESP research grants.

References

[1] Atmel Corporation, *8 bit AVR Microcontroller ATmega128(L) manual*, Atmel, (2004), 2467m-avr-11/04 edition.

[2] Atmel Corporation, *AVR Studio 4.14*, Atmel, (2005), available online at http://www.atmel.com/.

[3] Certicom Research, *SEC 1: elliptic curve cryptography*, (2000), available online at http://www.secg.org.

[4] H. Eberle, A. Wander, N. Gura, S. Chang-Shantz and V. Gupta, *Architectural extensions for elliptic curve cryptography over GF*(2^m) *on 8-bit microprocessors*, in "Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'05)," IEEE, (2005), 343–349.

[5] D. Estrin, R. Govindan, J. S. Heidemann and S. Kumar, *Next century challenges: scalable coordination in sensor networks*, in "Proceedings of Mobile Computing and Networking (MobiCom'99)," (1999), 263–270.

[6] R. Gallant, R. Lambert and S. Vanstone, *Faster point multiplication on elliptic curves with efficient endomorphisms*, in "Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'01)," Springer, (2001), 190–200.

[7] J. Großschädl, *TinySA: a security architecture for wireless sensor networks*, in "Proceedings of ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT'06)," ACM, (2006).

[8] N. Gura, A. Patel, A. Wander, H. Eberle and S. C. Shantz, *Comparing elliptic curve cryptography and RSA on 8-bit CPUs*, In "Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES'04)," Springer, (2004), 119–132.
[9] D. Hankerson, A. J. Menezes and S. Vanstone, "Guide to Elliptic Curve Cryptography," Springer-Verlag, Secaucus, 2003.
[10] J. L. Hill and D. E. Culler, *MICA: a wireless platform for deeply embedded networks*, IEEE Micro., **22** (2002), 12–24.
[11] A. Karatsuba and Y. Ofman, *Multiplication of many-digital numbers by automatic computers*, Transl. Physics-Doklady, **7** (1963), 595–596.
[12] A. Kargl, S. Pyka and H. Seuschek, *Fast arithmetic on ATmega128 for elliptic curve cryptography*, preprint, available online at http://eprint.iacr.org/2008/442.
[13] C. Karlof, N. Sastry and D. Wagner, *TinySec: a link layer security architecture for wireless sensor networks*, In "Proceedings of 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)", ACM, (2004), 162–175.
[14] N. Koblitz, *Elliptic curve cryptosystems*, Math. Comput., **48** (1987), 203–209.
[15] G.-J. Lay and H. G. Zimmer, *Constructing elliptic curves with given group order over large finite fields*, in "Algorithmic Number Theory," (1994), 250–263.
[16] C. H. Lim and P. J. Lee, *More flexible exponentiation with precomputation*, in "Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'01)," Springer, (1994), 95–107.
[17] J. López and R. Dahab, *Fast multiplication on elliptic curves over GF*(2^m) *without precomputation*, in "Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)," Springer, (1999), 316–327.
[18] J. López and R. Dahab, *Improved algorithms for elliptic curve arithmetic in GF*(2^n), in "Proceedings of Workshop on Selected Areas in Cryptography (SAC'98)," Springer, (1999), 201–212.
[19] J. López and R. Dahab, *High-speed software multiplication in GF(*2^m*)*, in "Proceedings of International Conference on Cryptology in India (INDOCRYPT'00)," Springer, (2000), 203–212.
[20] D. J. Malan, M. Welsh and M. D. Smith, *A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography*, in "Proceedings of IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON'04)," (2004).
[21] A. Menezes, T. Okamoto and S. Vanstone, *Reducing elliptic curve logarithms to logarithms in a finite field*, IEEE Trans. Inform. Theory, **39** (1993), 1639–1646.
[22] V. Miller, *Uses of elliptic curves in cryptography*, in "Advances in Cryptology (CRYPTO'85)," Springer, (1986), 417–426.
[23] L. B. Oliveira, D. F. Aranha, E. Morais, F. Daguano, J. López and R. Dahab, *TinyTate: computing the Tate pairing in resource-constrained sensor nodes*, in "Proceedings of IEEE International Symposium on Network Computing and Applications (NCA'07)," IEEE, (2007), 318–323.
[24] L. B. Oliveira, A. Kansal, B. Priyantha, M. Goraczko and F. Zhao, *Secure-TWS: authenticating node to multi-user communication in shared sensor networks*, in "Proceedings of International Conference on Information Processing in Sensor Networks (IPSN'09)," IEEE, (2009), 289–300.
[25] L. B. Oliveira, M. Scott, J. López and R. Dahab, *TinyPBC: pairings for authenticated identity-based non-interactive key distribution in sensor networks*, in "Proceedings of International Conference on Networked Sensing Systems (INSS'08)," IEEE, (2008), 173–180.
[26] A. Perrig, R. Szewczyk, V. Wen, D. Culler and J. D. Tygar, *SPINS: security protocols for sensor networks*, Wireless Networks, **8** (2002), 521–534.
[27] T. Satoh, B. Skjernaa and Y. Taguchi, *Fast computation of canonical lifts of elliptic curves and its application to point counting*, Finite Fields Appl., **9** (2003), 89–101.
[28] M. Scott, *MIRACL – multiprecision integer and rational arithmetic C/C++ library*, available online at http://www.shamus.ie/.
[29] S. C. Seo, D. Han and S. Hong, *TinyECCK: efficient elliptic curve cryptography implementation over GF*(2^m) *on 8-bit MICAz mote*, preprint, available online at http://eprint.iacr.org/2008/122.
[30] J. A. Solinas, *Efficient arithmetic on Koblitz curves*, Designs Codes Cryptogr., **19** (2000), 195–249.

[31] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier and R. Dahab, *NanoECC: testing the limits of elliptic curve cryptography in sensor networks*, in "Proceedings of European Conference on Wireless Sensor Networks (EWSN'08)," Springer, (2008), 305–320.
[32] L. Uhsadel, A. Poschmann and C. Paar, *Enabling full-size public-key algorithms on 8-bit sensor nodes*, in "Proceedings of European Workshop on Security in Ad-hoc and Sensor Networks (ESAS '07)," (2007), 73–86.
[33] H. Wang and Q. Li, *Efficient implementation of public key cryptosystems on mote sensors*, in "Proceedings of International Conference on Information and Communication Systems (ICICS'06)," Springer, (2006), 519–528.
[34] H. Yan and Z. J. Shi, *Studying software implementations of elliptic curve cryptography*, in "Proceedings of International Conference on Information Technology: New Generations (ITNG'06)," IEEE, (2006), 78–83.

Received June 2009; revised December 2009.

E-mail address: dfaranha@ic.unicamp.br
E-mail address: rdahab@ic.unicamp.br
E-mail address: jlopez@ic.unicamp.br
E-mail address: leob@ft.unicamp.br