Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x Family of Microcontrollers

Jorge Guajardo[1⋆], Rainer Blümel[2], Uwe Krieger[2], and Christof Paar[1]

[1] ECE Department, Worcester Polytechnic Institute, Worcester, MA 01609, USA
{guajardo,christof}@ece.wpi.edu
[2] cv cryptovision gmbh, Munscheidstr. 14, 45886 Gelsenkirchen, Germany
{Rainer.Bluemel,Uwe.Krieger}@cryptovision.com

In K. Kim (Ed.): PKC 2001, LNCS 1992, pp. 365-382, Korea, February 2001.

Abstract. This contribution describes a methodology used to efficiently implement elliptic curves (EC) over $GF(p)$ on the 16-bit TI MSP430x33x family of low-cost microcontrollers. We show that it is possible to implement EC cryptosystems in highly constrained embedded systems and still obtain acceptable performance at low cost. We modified the EC point addition and doubling formulae to reduce the number of intermediate variables while at the same time allowing for flexibility. We used a Generalized-Mersenne prime to implement the arithmetic in the underlying field. We take advantage of the special form of the moduli to minimize the number of precomputations needed to implement inversion via Fermat's Little theorem and the k-ary method of exponentiation. We apply these ideas to an implementation of an elliptic curve system over $GF(p)$, where $p = 2^{128} - 2^{97} - 1$. We show that a scalar point multiplication can be achieved in 3.4 seconds without any stored/precomputed values and the processor clocked at 1 MHz.

1 Introduction

It is widely recognized that data security will play a central role in the design of future IT systems. Until a few years ago, the PC had been the major driver of the digital economy. PCs have processors with large RAM memories and fast CPUs that make most cryptographic algorithms practical from a user's satisfaction point of view. Recently, however, there has been a shift towards IT applications realized as embedded systems. In fact, 98% of all microprocessors sold today are embedded in household appliances, vehicles, and machines on factory floors [12, 6]. Not only are embedded devices already ubiquitous in our lives, but it is predicted that in the very near future, we will be able to add to these devices two simple technologies: reliable wireless communication and sensing and actuation functions [6]. On the other hand, these new applications represent many challenges among which security and privacy will play an important role [6].

⋆ Part of this work was performed while the author was at cv cryptovision gmbh.

Embedded devices are very different from PCs from the computational resources and memory availability point of view. Generally, embedded computers possess CPUs with very slow clock rates and a relatively small pool of memory. In addition, embedded systems are usually designed to consume small amounts of energy. Despite these constraints, we want to be able to run the same (or similar) types of applications that we run today in a fast computer. These applications often need to talk to each other and transmit information over wireless channels which are insecure by nature. Thus, cryptographic algorithms, which are computationally intensive by design, are imperative for embedded applications. In particular, it is important to show that it is feasible to implement cryptographic algorithms in constrained environments and, at the same time, be able to obtain acceptable levels of performance.

Our contribution deals with the implementation of Elliptic Curve (EC) cryptosystems [25, 20] on the TI MSP430x33x family of devices. These 16-bit microcontrollers are one example of embedded device used for extremely low-power and low-cost applications, running at a maximum frequency of 3.8 MHz. Elliptic curves, on the other hand, are a particularly attractive option because of their relatively short operand length as compared to RSA and systems based on the discrete logarithm (DL) in finite fields.

The remaining of this contribution is organized as follows. Section 2 gives a survey of previous implementations of public-key algorithms on embedded processors. Section 3 describes the choice of parameters used in our EC implementation and a modification to the point addition and doubling algorithms which allow for a reduction in the memory requirements. The architecture of the TI MSP430x33x family of devices is covered in Section 4. In Section 5, we modify the k-ary algorithm to take advantage of the special form of the moduli and compute the inverse of an element in $GF(p)$ via Fermat's Little theorem. We also describe ways to tailor multiplication, squaring, and modular reduction algorithms to the architecture of the processor, thus, making the algorithms more efficient. Finally, Sections 6 and 7 summarize our implementation results and provide recommendations for possible enhancements.

2 Previous Work

Most of the cryptographic research conducted to date has been independent of hardware platforms, and little research has focused on algorithm optimization for specific processors. In the following, we will review previous implementations of public-key algorithms on embedded processors.

In [5], the Barret modular reduction method is introduced. The author implemented RSA on the TI TMS32010 DSP. A 512-bit RSA exponentiation took on the average 2.6 seconds running at the DSP's maximum speed of 20 MHz. Reference [11] describes the implementation of a cryptographic library designed for the Motorola DSP56000 which was clocked at 20 MHz. The authors focused on the integration of modular reduction and multi-precision multiplication according to Montgomery's method [26, 7]. This RSA implementation achieved a data

rate of 11.6 Kbits/s for a 512-bit exponentiation using the Chinese Remainder Theorem (CRT) and 4.6 Kbits/s without using it.

The authors in [16] described an ECDSA implementation over $GF(p)$ on the M16C, a 16-bit 10 MHz microcomputer. Reference [16] proposes the use of a field of prime characteristic $p = e2^c \pm 1$, where e is an integer within the machine word size and c is a multiple of the machine word size. This choice of field allows to implement multiplication in $GF(p)$ in a small amount of memory. Notice that [16] uses a randomly generated curve with the a coefficient of the elliptic curve equal to $p - 3$. This reduces the number of operations needed for an EC point doubling. They also modify the point addition algorithm in [29] to reduce the number of temporary variables from 4 to 2. This contribution uses a 31-entry table of precomputed points to generate an ECDSA signature in 150 msec. On the other hand, scalar multiplication of a random point takes 480 msec and ECDSA verification 630 msec. The whole implementation occupied 4 Kbyte of code/data space.

In [17], two new methods for implementing public-key cryptography algorithms on the 200 MHz TI TMS320C6201 DSP are proposed. The first method is a modified implementation of the Montgomery variant known as the Finely Integrated Operand Scanning (FIOS) algorithm [7] suitable for pipelining. The second approach suggests a method for reducing the number of multiplications and additions used to compute $2^m P$, for P a point on the elliptic curve and m some integer. The final code implemented RSA and DSA combined with the k-ary method for exponentiation, and ECDSA combined with the improved method for multiple point doublings, sliding window exponentiation, and signed binary exponent recoding. The total instruction code was 41.1 Kbytes. They achieved 11.7 msec for a 1024-bit RSA signature using the CRT (1.2 msec for verification assuming a 17-bit exponent) and 1.67 msec for a 192-bit ECDSA signature over $GF(p)$ (6.28 msec for verification and 4.64 msec for general point multiplication).

Recently, two papers have introduced fast implementations on 8-bit processors over Optimal Extension Fields (OEFs), originally introduced in [3]. Reference [9] reports on an ECC implementation over the field $GF(p^m)$ with $p = 2^{16} - 165$, $m = 10$, and $f(x) = x^{10} - 2$ is the irreducible polynomial. The authors use the column major multiplication method for field multiplication and squaring, for the specific case in which $f(x)$ is a binomial. They achieve better performance than when using Karatsuba multiplication because in this processor additions and multiplications take the same number of cycles. Modular reduction is done through repeated use of the division step instruction. For inversion, they use the variant of the Itoh and Tsujii Algorithm [18] proposed in [4]. For EC arithmetic they combine the mixed coordinate system methods of [10] and [22]. These combined methods allow them to achieve 122 msec for a 160-bit point multiplication on the CalmRISC with MAC2424 math coprocessor running at 20 MHz. The second paper [34] describes a smart card implementation over the field $GF((2^8 - 17)^{17})$ without the use of a coprocessor.

Reference [34] focuses on the implementation of ECC on the 8051 family of microcontrollers, popular in smart cards. The authors compare three types of fields: binary fields $GF(2^k)$, composite fields $GF((2^n)^m)$, and OEFs. Based on multiplication timings, the authors conclude that OEFs are particularly well suited for this architecture. A key idea of this contribution is to allow each of the 16 most significant coefficients resulting from a polynomial multiplication to accumulate over three 8-bit words instead of reducing modulo $p = 2^8 - 17$ after each 8-bit by 8-bit multiplication. Fast field multiplication allows the implementation to have relatively fast inversion operations following the method proposed in [4]. This, in turn, allows for the use of affine coordinates for point representation. Finally, the authors combine the methods above with a table of 9 precomputed points to achieve 1.95 sec for a 134-bit fixed point multiplication and 8.37 sec for a general point multiplication using the binary method of exponentiation.

3 Elliptic Curves over $GF(p)$

In this paper, we will only be concerned with non-supersingular elliptic curves over $GF(p)$, $p > 3$. Thus, an elliptic curve E will be defined to be the set of points $P = (x, y)$ with $x, y \in GF(p)$ and satisfying the cubic equation $y^2 = x^3 + ax^2 + b$, where $a, b \in GF(p)$ with $4a^2 + 27b^2 \neq 0 \pmod{p}$, together with the point at infinity $\mathcal{O}$. The points (x, y) form an abelian group under "addition" where the group operation is defined as in [23]. It is a well known fact that in *affine* representation, one needs to compute the inverse of an element in $GF(p)$ to perform an addition or a doubling of a point $P \in E$. Inversion can be a very time consuming operation (when compared to multiplication, addition, and subtraction in the finite field) and thus, to avoid inversion in the group operation, one can represent points in *projective* coordinates. Given a point $P = (x, y)$ in affine coordinates, one obtains the projective coordinate representation of $P = (X, Y, Z)$ by:

$$X = x; \;\; Y = y; \;\; Z = 1 \tag{1}$$

On the other hand, the projective coordinates of a point are not unique and they require more bandwidth. Thus, for transmission/exchange of data, affine coordinate representation is the method of choice. Given a point $P = (X, Y, Z)$ in projective coordinates, the corresponding affine coordinate representation of $P = (x, y)$ is given by:

$$x = \frac{X}{Z^2} \;\; y = \frac{Y}{Z^3} \tag{2}$$

where we have chosen the Jacobian representation [8, 10].

3.1 Addition and Doubling Formulae in Jacobian Representation

Using the representation in (1) and (2) is equivalent to using a curve equation of the form:

$$E: \quad Y^2 = X^3 + aXZ^4 + bZ^6 \tag{3}$$

Then, one can define addition and doubling of points as follows. Let $P_0 = (X_0, Y_0, Z_0), P_1 = (X_1, Y_1, Z_1), P_2 = (X_2, Y_2, Z_2) \in E$, then if $P_0 = P_1$:

$$P_2 = 2P_1 = \begin{cases} X_2 = M^2 - 2S \\ Y_2 = M(S - X_2) - T \\ Z_2 = 2Y_1 Z_1 \end{cases} \tag{4}$$

where $M = 3X_1^2 + aZ_1^2$, $S = 4X_1Y_1^2$, and $T = 8Y_1^4$. On the other hand, if $P_0 \neq P_1$:

$$P_2 = P_0 + P1 = \begin{cases} X_2 = R^2 - TW \\ 2Y_2 = VR - MW^3 \\ Z_2 = Z_0 Z_1 W \end{cases} \tag{5}$$

where $W = X_0Z_1^2 - X_1Z_0^2$, $R = Y_0Z_1^3 - Y_1Z_0^3$, $T = X_0Z_1^2 + X_1Z_0^2$, $M = Y_0Z_1^3 + Y_1Z_0^3$, and $V = TW^2 - 2X_2$.

Based on (4) and (5) one can implement doubling of EC points using 5 temporary variables and addition of EC points using 7 temporary variables [29]. The number of temporary variables used in the addition and doubling operation can further be reduced to 2 temporary variables and 3 output variables if one follows the ideas proposed in [16].

3.2 Elliptic Curve Arithmetic Implementation

Our point addition and doubling routines follow closely the formulae in [29] combined with modifications similar to the ones proposed in [16]. In particular, we follow a similar idea to minimize the number of temporary variables used in performing a point addition or a point doubling. Our implementation requires 5 temporary variables but it also allows for greater flexibility. We perform the following computation $P_2 \leftarrow P_0 + P_1$ as opposed to $P_0 \leftarrow P_0 + P_1$ as described in [16]. Another difference is that whenever we have to multiply by 2,3, or 8 in the algorithms, we substitute the multiplication by one, two or three additions respectively. As it will be seen in Section 6 one modular multiplication time is about 10 modular addition times, thus it makes sense to exchange multiplications for additions whenever possible. Similarly, we have used a special squaring routine whenever possible since squaring is 24% more efficient than regular multiplication. Finally, notice that because point addition and doubling will never occur simultaneously, it is possible to use the temporary memory space available for the point addition routine in the point doubling routine, effectively reducing the memory required by a factor of 2.

For scalar point multiplication, which is the most important operation in ECDSA [2, 27] signature generation or verification operation we implemented the binary method for exponentiation [19] which is simple to implement and minimizes memory requirements. It is important to point out that signature generation times can be further improved by using point precomputation since the base point in ECDSA is a system parameter. However, this method has the drawback of increasing the memory required for the implementation.

3.3 Elliptic Curve Parameters

In this contribution, we consider two 128-bit elliptic curves, both specified in [31]. Notice that the set of parameters recommended in [31] is also identical to the set of parameters recommended in [1]. Both curves presented in this section are defined over $GF(p)$ and their parameters are verifiable generated at random. The parameters are a sextuple $T = (p, a, b, n, h, G)$, where a, b are the elliptic curve coefficients as defined in (3), G is a base point of prime order n and represented in affine coordinates, and h is the cofactor $\#E(GF(p))/n$, where $\#E(GF(p))$ denotes the number of points in the curve E. We also include the seed S used to choose E according to [2].

Parameters secp128r1.

$$
\begin{aligned}
p &= (\text{FFFFFFFD FFFFFFFF FFFFFFFF FFFFFFFF})_{16} = 2^{128} - 2^{97} - 1 \\
a &= (\text{FFFFFFFD FFFFFFFF FFFFFFFF FFFFFFFC})_{16} = p - 3 \\
b &= (\text{E87579C1 1079F43D D824993C 2CEE5ED3})_{16} \\
n &= (\text{FFFFFFFE 00000000 75A30D1B 9038A115})_{16} \\
h &= (01)_{16} \\
G &= ((\text{161FF752 8B899B2D 0C28607C A52C5B86})_{16}, \\
&\quad (\text{CF5AC839 5BAFEB13 C02DA292 DDED7A83})_{16}) = (x, y) \\
S &= (\text{000E0D4D 696E6768 75615175 0CC03A44 73D03679})_{16}
\end{aligned}
$$

Parameters secp128r2.

$$
\begin{aligned}
p &= (\text{FFFFFFFD FFFFFFFF FFFFFFFF FFFFFFFF})_{16} = 2^{128} - 2^{97} - 1 \\
a &= (\text{D6031998 D1B3BBFE BF590C9B BFF9AEE1})_{16} \\
b &= (\text{5EEEFCA3 80D02919 DC2C6558 BB6D8A5D})_{16} \\
n &= (\text{3FFFFFFF 7FFFFFFF BE002472 0613B5A3})_{16} \\
h &= (04)_{16} \\
G &= ((\text{7B6AA5D8 5E572983 E6FB32A7 CDEBC140})_{16}, \\
&\quad (\text{27B6916A 894D3AEE 7106FE80 5FC34B44})_{16}) = (x, y) \\
S &= (\text{004D696E 67687561 517512D8 F03431FC E63B88F4})_{16}
\end{aligned}
$$

4 The TI MSP430x33x Family of Microcontrollers

The TI MSP430x33x is a 16-bit RISC based family of microcontrollers with a 16-by-16, 16-by-8, 8-by-16, and 8-by-8 bit hardware multiplier. This family of devices is commonly used in low-cost and low-power applications involving electronic gas, water, and electric meters and other sensor systems that capture

analog signals, convert them to digital values, and then process, display, or transmit the data to a host system. It is important to point out that they have been specially designed for ultra-low power applications. Family members include the MSP430C336 with 24 Kbytes of ROM and the MSP430C337, MSP430P337A, MSP430P337A and PMS430E337A with 32 Kbytes of ROM (or EPROM/OTP). All of the MPS430x33x family members include 1 Kbyte of on-chip RAM and can be clocked to a maximum frequency of 3.8 MHz. The total addressable space is 64 Kbytes [32].

Instruction fetches from program memory (ROM) are always 16-bit access, whereas data memory can be accessed using 16-bit (word) or 8-bit (byte) instructions. In addition to program code, data can also be placed in the ROM area of the memory map and it can be accessed using word or byte instructions. This is useful for storing data tables, for example. At the top of the 64 kilobytes of addressable space 16 words of memory are reserved for the interrupt and reset vectors. The remaining address space (after ROM, RAM, and interrupts) is used for peripherals.

The architecture of the MSP430 family is based on a memory-to-memory architecture, a common address space for all functional blocks and a reduced instruction set. The MSP430 RISC CPU includes sixteen 16-bit registers $R0-R15$. Registers $R0-R3$ are special function registers or SFRs and have the dedicated functions of Program Counter, Status Register, Constant Generator, and Stack Pointer. The remaining registers ($R4-R15$) are general purpose registers and have no restrictions in their usage. All registers except for the Constant Generator can be accessed using the complete instruction set. This includes 27 core instructions and 24 additional emulated instructions (instructions that make programming simpler and that are substituted for core instructions by the assembler). All instructions are single or double operand instructions. The MSP430 devices support 7 different addressing modes. They include: register mode, indexed mode, symbolic mode, absolute mode, indirect register mode, indirect auto-increment, and immediate mode. Depending on the addressing mode the instructions take between 1 cycle and 6 cycles to execute. As a final remark, notice that the result of a multiply instruction is available one clock cycle after loading the two operands into the hardware multiplier [33].

5 Finite Field Arithmetic

In this section we summarize the rational behind some of our design choices. We emphasize the description of a new inversion algorithm specially suited for primes of special form. We also describe how our implementation was tailored to take the most advantage of the architecture of the MSP430 family of processors.

5.1 Modular Reduction

Reduction Modulo $p = 2^{128} - 2^{97} - 1$. One of the critical operations when implementing finite field arithmetic is modular reduction. We chose the

field $GF(p)$ where $p = 2^{128} - 2^{97} - 1$, for the underlying arithmetic of our EC implementation. The first thing to notice is that p is a generalized Mersenne prime and that this type of fields allow for efficient reduction as described in [28]. Following [28], we first notice that any number $A \in GF(p)$ such that $A < p^2$ can be written as:

$$A = \sum_{i=0}^{i=15} a_i 2^{16i} \quad 0 \leq a_i \leq 2^{16} - 1 \tag{6}$$

where we have chosen $2^{16} - 1$ to be the maximum digit value because of the MSP430 16-bit based architecture. Then, it is easy to see that only the 8 most significant digits a_i need to be modulo reduced. Using the fact that $p = 2^{128} - 2^{97} - 1$, one obtains the following identities:

$$a_8 2^{128} \equiv 2a_8 2^{96} + a_8 \pmod{p} \tag{7}$$
$$a_9 2^{144} \equiv 2a_9 2^{112} + a_9 2^{16} \pmod{p} \tag{8}$$
$$a_{10} 2^{160} \equiv 4a_{10} 2^{96} + a_{10} 2^{32} + 2a_{10} \pmod{p} \tag{9}$$
$$a_{11} 2^{176} \equiv 4a_{11} 2^{112} + a_{11} 2^{48} + 2a_{11} 2^{16} \pmod{p} \tag{10}$$
$$a_{12} 2^{192} \equiv 8a_{12} 2^{96} + a_{12} 2^{64} + 2a_{12} 2^{32} + 4a_{12} \pmod{p} \tag{11}$$
$$a_{13} 2^{208} \equiv 8a_{13} 2^{112} + a_{13} 2^{80} + 2a_{13} 2^{48} + 4a_{13} 2^{16} \pmod{p} \tag{12}$$
$$a_{14} 2^{224} \equiv (16+1)a_{14} 2^{96} + 2a_{14} 2^{64} + 4a_{14} 2^{32} + 8a_{14} \pmod{p} \tag{13}$$
$$a_{15} 2^{240} \equiv (16+1)a_{15} 2^{112} + 2a_{15} 2^{80} + 4a_{15} 2^{48} + 8a_{15} 2^{16} \pmod{p} \tag{14}$$

Adding up the first 8 words from A, relations (7) through (14), and reducing modulo p, one readily obtains $A \bmod p$. Notice that one only needs single precision additions to perform this operation. In general, whenever a product of the form $2^i a_j$ happens in relations (7) through (14), it is more efficient to compute the product and then add it to the partial result than adding a_j $(2^i - 1)$-times. Notice that the final result after adding relations (7) through (14) and the first 8 words of A, only needs nine 16-bit words to be represented and that one will need at most 2 subtractions to reduce this result modulo p. Thus, we kept all the partial sums in 9 of the 12 general purpose registers to minimize the fetches to memory during the modular reduction operation.

Modular Reduction for Arbitrary *p*. In both ECDSA signature and ECDSA verification operations, one needs to perform modular reductions modulo the order of the base point G. In general, $\text{ord}(G) = n$ is not of special form and, thus, one needs to implement modular reduction for arbitrary moduli. We used Montgomery modular reduction [26]. In particular, we implemented the Separated Operand Scanning (SOS) method as proposed in [7]. Algorithm 1 summarizes the SOS method for Montgomery reduction.

Algorithm 1 SOS Method of Montgomery Reduction

INPUT: $t = \overline{a} \cdot \overline{b} = (a \cdot r \bmod n) \cdot (b \cdot r \bmod n) = (t_{2s-1}, \ldots, t_0)$
$n = (n_{s-1}, \ldots, n_0)$
$r = 2^{sw}$ and w is typically the word size of the processor
n'_0 where $n' = (n'_{s-1}, \ldots, n'_0)$, satisfies $(r)(r^{-1}) + (-n)(n') = 1$
OUTPUT: $\overline{c} = (\overline{c}_{s-1}, \ldots, \overline{c}_0) = a \cdot b \cdot r \bmod n$

01 for $i = 0$ to $s - 1$
02 $c = 0$
03 $m = t_i \cdot n'_0 \bmod 2^w$
04 for $j = 0$ to $s - 1$
05 $(c, t_{i+j}) = t_{i+j} + m \cdot n_j + c$
06 endfor
07 while $(c \neq 0)$
08 $(c, t_{i+s}) = t_{i+s} + c$
09 $i = i + 1$
10 endwhile
11 endfor
12
13 for $j = 0$ to s
14 $u_j = t_{j+s}$
15 endfor
16
17 if $u \geq n$
18 return $\overline{c} = u - n$
19 else
20 *return* $\overline{c} = u$
21 endif

First, we notice that in our particular implementation $s = 128/16 = 8$, thus it is possible to load all of n into 8 of the 12 general purpose registers. This simple observation saves 2 cycles per iteration when loading n_j in line *05* of Algorithm 1 into the multiplier. Since the j-loop executes s-times and this in turn is executed s-times by the i-loop, it gives a total of $8 \cdot 8 \cdot 2 = 128$ cycles in savings. In addition, since m stays constant within the j-loop, we can load m into the multiplier before starting the execution of the j-loop and only load n_j each time we execute line *05* in Algorithm 1. This observation saves $4 \cdot 8 \cdot 7 = 224$ cycles.

5.2 Multiplication and Squaring

Multiplication and squaring operations for long number arithmetic are described in [24]. For completeness Algorithm 2 summarizes the school-book method for multiplication. We notice Algorithm 2 has a similar structure to that of Algorithm 1. In particular, in line *04*, for a fixed word a_i we compute s inner products. Thus, the same optimizations that were applied to Algorithm 1 are applicable to Algorithm 2. Finally, we notice that a squaring operation is 24% cheaper to

compute than a regular multiplication because, in the squaring case, we do not need to fetch from memory the words from the second operand.

Algorithm 2 School-book Method for Multiplication

INPUT: $A = (a_{s-1}, \ldots, a_0)$
$B = (b_{s-1}, \ldots, b_0)$
OUTPUT: $C = (c_{2s-1}, \ldots, c_0) = a \cdot b$

01 for $i = 0$ to $s - 1$
02 $c = 0$
03 for $j = 0$ to $s - 1$
04 $(c, t_{i+j}) = t_{i+j} + a_i \cdot b_j + c$
05 endfor
06 $t_{i+s} = c$
07 endfor

5.3 Modular Addition and Subtraction

We followed the methods described in [24] to implement modular addition and modular subtraction. Given two elements $A, B \in GF(p)$, $C = A + B \bmod p$ can be obtained by first adding A and B and reducing modulo p. This last step can be accomplished by simply subtracting p from the partial result $A + B$, rather than using the method described in Section 5.1. The same comments apply to modular subtraction.

5.4 A New Inversion Algorithm for Moduli of Special Form

Several methods to compute the inverse of an element in $GF(p)$ exist. They include methods based on the extended Euclidean algorithm [19] such as the binary Euclidean algorithm and the Almost Inverse algorithm [30], methods based on Itoh and Tsujii's inversion algorithm and its variants [18, 14, 4], and methods based on Fermat's Little theorem. Despite the fact that on the average Fermat based inversion is slower than methods based on the Euclidean algorithm, it has several advantages. First, Fermat based inversion is easier to implement and it allows for implementations that occupy less code space than those implementations based on Euclid's algorithm. Second, in an ECC implementation that uses projective coordinates, inversion is not time critical.

The new inversion algorithm is based on Fermat's Little theorem, i.e., on the observation that for any non-zero element $A \in GF(p)$, $A^{-1} \equiv A^{p-2} \bmod p$. In particular, the algorithm that we are proposing in this section is only applicable to the computation of inverses when p is a Mersenne or Generalized-Mersenne prime. Despite this apparent constraint on the applicability of the algorithm, it is our opinion that the algorithm is highly relevant given the recent parameter recommendations by NIST [28] and SECG [31], both of which include Generalized-Mersenne primes. The basic idea of the new algorithm is to minimize

the number of precomputations in the k-ary method for exponentiation [24, 13] used to compute the inverse via Fermat's Little theorem. This is possible because of the special form of Mersenne and Generalized-Mersenne primes and, as a consequence, of $p-2$. Algorithm 3 describes the k-ary method for exponentiation.

Algorithm 3 k-ary Method for Exponentiation

INPUT: $A \in GF(p)$
$e = (e_s e_{s-1} \dots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$
OUTPUT: $C = A^e$

01 Precomputation
02 $A_0 = 1$
03 for $i = 1$ to $(2^k - 1)$
04 $A_i = A_{i-1} \cdot A$ (Thus, $A_i = A^i$)
05 endfor
06 $A = 1$
07 for $i = s$ down to *0*
08 $A = A^{2^k}$
09 $A = A \cdot A_{e_i}$
10 endfor
11 $C = A$

Notice that given a window size k, the precomputation stage of Algorithm 3 computes all the possible values A^i for $i = 1, \dots, 2^k - 1$ (observe that the Improved k-ary algorithm can reduce the number of precomputed values in half [24]). These values are then used in line *09* of the k-ary algorithm. In addition, the number of precomputed values (i.e., k) is determined by two factors: the amount of RAM memory (or the size of the cache) available in the processor and the number of operations (multiplications) used to compute the table values . In particular, the table of precomputed values should fit in RAM memory (or in the cache), if we want to ensure that memory accesses to the table are fast. Reference [13] also gives the following complexity formula for the number of multiplications (assuming squarings and multiplications take the same time) performed in the k-ary method, in the worst case:

$$\#\text{Multiplications} = 2^k - 2 + \left(1 + \frac{1}{k}\right) \lfloor \log_2 e \rfloor \tag{15}$$

Equation (15) gives an easy way to find the optimum value of k for a given exponent size. In particular, $k = 4$ minimizes (15) for a 128-bit exponent. We also notice that in general not all the precomputed values will be used. In particular, only the values A^{e_i} which correspond to the e_i digits happening in the exponent will be used. For the case $k = 4$ these values correspond to the hexadecimal digits present in the exponent when e is in radix-16 representation. The above discussion leads us to believe that it would be of great help if we find a way to reduce the number of precomputations performed in the k-ary method.

Next, we turn our attention to the exponent. Notice, that to compute the inverse of any element in $GF(p)$ we need to raise to the $p - 2$ power, where p

is a Mersenne prime or Generalized-Mersenne prime. Thus, our exponent has a very special form. We first consider the case in which $p = 2^r - 1$.

Theorem 1. *Let $p = 2^r - 1$ be a prime. Then, one can compute the inverse of an element $A \in GF(p)$ using Algorithm 3, with only two precomputed values.*

Proof. If $p = 2^r - 1$, then $p - 2 = 0xFF \ldots FFD$ in hexadecimal representation. But, the hexadecimal representation of $p-2$ (the exponent) corresponds exactly to the values that will be used in line *09* of Algorithm 3, which for the case $k = 4$ are A^{13} and A^{15}. Thus, we only need to precompute and store 2 values. □

For the Generalized-Mersenne prime used here the case is very similar.

Theorem 2. *Let $p = 2^r - 2^t - 1$ be a prime, with $r = 4 \cdot d$ for some d. Then, one can compute the inverse of an element $A \in GF(p)$ using Algorithm 3, with at most 3 precomputed values.*

Proof. If $p = 2^r - 2^t - 1$ with $r = 4 \cdot d$ then we can rewrite it as $p = 2^r - 2^u \cdot 2^{4 \cdot v} - 1$ where $u = 0, 1, 2, 3$, $t = u + 4 \cdot v$, and v is any positive integer less than $d - 1$. Notice that the value $2^r - 1$ can be written as $\sum_{i=0}^{i=r/4-1} \left(2^4 - 1\right) 2^{4 \cdot i}$ Then, it follows that we can write $p - 2$ as:

$$\left(2^4 - 1\right) 2^{r-4} + \left(2^4 - 1\right) 2^{r-8} + \cdots + \left(2^4 - 2^u - 1\right) 2^{4 \cdot v} + \left(2^4 - 1\right) 2^{4 \cdot v - 4} + \cdots + \left(2^4 - 1\right) 2^4 + \left(2^4 - 3\right)$$

Looking at the coefficients of the powers of 2^4, we see that there are three: $2^4 - 1$, $2^4 - 2^u - 1$, and $2^4 - 3$. This ends the proof. □

Notice, that Theorems 1 and 2 suggest an improved algorithm to compute the inverse of a non-zero element $A \in GF(p)$. Algorithm 4 describes the new algorithm.

Algorithm 4 Inversion Algorithm for Mersenne and Generalized-Mersenne Primes

INPUT: $A \in GF(p)$ with $p = 2^r - 1$ (Mersenne) or
$p = 2^r - 2^u 2^{4 \cdot v} - 1$ (Generalized-Mersenne prime from Theorem 2)
$e = (e_s e_{s-1} \ldots e_1 e_0)_b = p - 2$, with $b = 2^4$ and $0 \leq e_i \leq 2^4 - 1$
OUTPUT: $C = A^{-1} = A^e$

01 Precomputation
02 if $p = 2^r - 1$
02 $A_{13} = A^{13}$
03 $A_{15} = A^{15}$
04 if $p = 2^r - 2^u \cdot 2^{4 \cdot v} - 1$
05 $A_{2^4 - 2^u - 1} = A^{2^4 - 2^u - 1}$
06 $A_{13} = A^{13}$
07 $A_{15} = A^{15}$
08 $A = 1$
09 for $i = s$ down to *0*
10 $A = A^{2^4}$
11 $A = A \cdot A_{e_i}$
12 endfor
13 $C = A$

5.5 A Word about the Security of a 128-bit EC Implementation

In recent work, Lenstra and Verheul show that under particular assumptions, 952-bit RSA and DSS systems may be considered to be of equivalent security to 132-bit ECC systems [21]. The authors further argue that 132-bit ECC keys are adequate for commercial security in the year 2000. This notion of commercial security is based on the hypothesis that a 56-bit block cipher offered adequate security in 1982 for commercial applications.

This estimate has more recently been confirmed by the breaking of the ECC2K-108 challenge [15]. First, note that the field $GF(2^{128} - 2^{97} - 1)$ has an order of about 2^{128}. Breaking the Koblitz (or anomalous) curve cryptosystem over $GF(2^{108})$ required slightly more effort than a brute force attack against DES. Hence, an ECC over a 128-bit field which does not use a subfield curve is by a factor of $\sqrt{108} \cdot \sqrt{2^{20}} \approx 10000$ harder to break than the ECC2K-108 challenge or DES. Thus, based on current knowledge of EC attacks, the system proposed here is roughly security equivalent to a 69-bit block cipher. This implies that an attack would require about 10000 times as much effort as breaking DES. Note also that factoring the 512-bit RSA challenge took only about 2% of the time required to break DES or the ECC2K-108 challenge. This implies that an ECC over the proposed field $GF(2^{128} - 2^{97} - 1)$ offers far more security than the 512-bit RSA system which has been popular, for example, for fielded smart card applications. We would also like to point out that due to the shorter size of the operands (128 bits vs. 160 bits) one could potentially attack a signature scheme by trying to find collisions in the hash function. Nevertheless, we feel that our selection of field order provides medium-term security which is sufficient for many applications intended for the MSP430x33x family of microcontrollers.

6 Implementation and Results

6.1 Software Architecture

The EC arithmetic library for the MSP430x33x devices was designed with modularity in mind. Thus, the design consists of three levels as depicted in Figure 1. Level 1 includes basic arithmetic functions such as addition, subtraction, multiplication, and squaring routines; modulo arithmetic functions such as modular reduction for $p = 2^{128} - 2^{97} - 1$, addition and subtraction modulo p, Montgomery reduction; and other support routines such as memory copying and setting routines, long number comparisons (>,<,=), and shift-right operations. It is important to point out that many of the modular arithmetic routines were optimized for the prime that we chose. Level 2 consists of macros. The assembly language for the TI MSP430 allows the programmer to use macros which are substituted by the compiler at compile time. This enables the programmer to write the Level 3 routines in terms of the macros from Level 2, thus, making the elliptic curve arithmetic routines independent of the chosen field. In the future, this will allow us to change the underlying arithmetic without changing the top level routines. It is important to point that this could have been accomplished

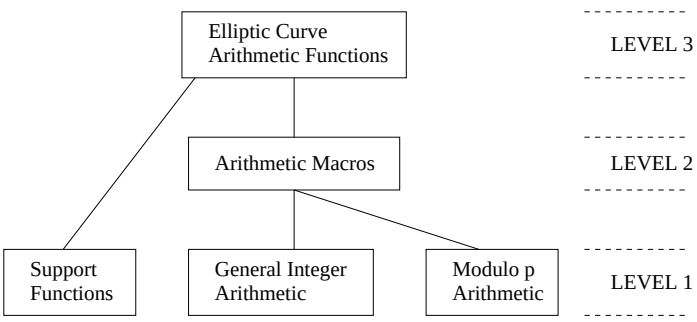

Fig. 1. Library Architecture

by using functions in Level 2 but this would increase the overhead (5 cycles per function call) which would impact negatively on the performance of our implementation. Finally, Level 3 routines included addition and doubling of elliptic curve points and scalar point multiplication.

6.2 Timings

The critical functions of the elliptic curve library were timed using the MSP430 TI Simulator version 2.30. The code was compiled using the MSP430 TI Macro Assembler version 1.08 and the MSP430 COFF TI Linker version 1.01. These tools are all part of the TI MSP-EVK430S330 Evaluation Kit which includes the PMS430E337HFD (UV EPROM) chip. Arithmetic is always assumed to be for 128-bit long operands. The actual timings are calculated for two frequencies: 1 MHz which seems to be a commonly used frequency in applications and 3 MHz which is close to the highest frequency that the MSP430 device can be clocked at. Table 1 summarizes the timings corresponding to basic arithmetic operations.

Some of the times reported are exact times. These include the execution times for multiplication and squaring Montgomery reduction. On the other hand, the times reported for modular addition, modular subtraction, and inversion modulo $p = 2^{128} - 2^{97} - 1$ using the modified k-ary method were computed by averaging out the worst and best times observed during the execution of these routines. Modular addition and modular subtraction, for example, perform the modular reduction by subtracting p from the intermediate result. In some cases, one can obtain the final result with only one subtraction, other times two subtractions are necessary. Best and worst case timings correspond to one or two subtractions performed in the reduction step, respectively. The remaining timings in Table 1 are the result of performing several computations with a given routine (Montgomery reduction, Montgomery exponentiation, etc.) and then averaging over the total number of operations performed. Finally, notice that the timings do not include the loading of the operands onto the registers where the routines expect their inputs. However, this operation can at most take 6 cycles for routines with 3 operands, so its impact on the overall performance

Table 1. Timings for basic arithmetic operations with 128-bit long operands.

Arithmetic Operation	Average Timing @ 1 MHz(usec)	Average Timing @ 3 MHz(usec)
addition modulo $p = 2^{128} - 2^{97} - 1$	164	55
subtraction modulo $p = 2^{128} - 2^{97} - 1$	156	52
multiplication	1425	475
squaring	998	333
reduction modulo $p = 2^{128} - 2^{97} - 1$	343	114
Montgomery reduction	1626	542
inversion modulo $p = 2^{128} - 2^{97} - 1$ via modified k-ary method.	235.9 msec	78.6 msec

of an elliptic curve operation is negligible. Table 2 presents the timings for the elliptic curve operations. The point doubling timings depend on whether the coefficient a of the elliptic curve is equal to $p-3$ or not. So both timings have been included. The longer timing corresponds to the case $a \neq p-3$. The point addition timings depend on whether the Z coordinate of the second point is equal to 1 or not. If it is equal to one then the number of operations is smaller and, thus, the addition operation takes less time. In this case we have also included both timings. The fulladd operation (a wrapper routine that can perform both point addition and point doubling) if used only for adding ($P0 \neq P1$) takes less time than when a doubling ($P0 = P1$) is performed. The reason for this is that fulladd first performs an add and if the output is $P0 + P1 = (0, 0, 0)$ (which is not a point on the elliptic curve but rather a pointer indicating that the doubling routine must be used), it performs a point doubling. Only timings for the case $P0 \neq P1$ are included. It is important to point out that this is the case of most relevance in practice since one can always make the Z coordinate of the second operand equal to 1. In fact, it should not occur that we perform a double with a fulladd when implementing point multiplication using the binary, k-ary, sliding window, or addition-subtraction algorithms since the point G is usually of prime order and in general the multiplier is chosen to be less than the order of G. In all cases, it makes no sense to average the two values because the timings correspond to situations which will not occur at the same time or in the same implementation. For example, one either chooses a curve with the a coefficient equal to $p-3$ or one does not. Finally, the timings for 128-bit point multiplication were computed using a 128-bit long exponent.

7 Conclusions

In this contribution, we have described a practical implementation of an EC cryptosystem over a prime field, where the prime is a Generalized-Mersenne prime, on the TI MSP430x33x family of low-cost and low-power microprocessors. We show how the special form of the Generalized-Mersenne prime can be used to

Table 2. Timings for elliptic curve operations assuming a 128-bit base field.

Elliptic Curve Operation	Average Timing @ 1 MHz (msec)	Average Timing @ 3 MHz (msec)
point doubling $a = p - 3$	15.1	5.0
point doubling $a \neq p - 3$	17.7	5.9
point addition $Z = 1$	20.7	6.9
point addition $Z \neq 1$	29.1	9.6
point fulladd $(P0 \neq P1)$ $Z = 1$	21.1	7.0
point fulladd $(P0 \neq P1)$ $Z \neq 1$	29.5	9.8
point subtraction $(P0 \neq P1)$ $Z = 1$	21.2	7.2
point multiplication using the binary method of exponentiation $(a = p - 3)$	3.4 sec	1.1 sec
point multiplication using the binary method of exponentiation $(a \neq p - 3)$	3.8 sec	1.3 sec

implement a new inversion algorithm, based on Fermat's Little theorem and the k-ary method for exponentiation, which minimizes the number of precomputed values, thus, also minimizing the memory requirements of the inversion operation. We would like to point out that even though $k = 4$ minimizes (15), this value might not be the optimum for our algorithm. Since we only required two (or three) precomputed values it is possible to increase the size of k, thus making the algorithm more efficient. In fact, it is easy to verify that a value of $k = 16$ will reduce in half the number of multiplications required to perform the inversion.

In addition, we tailor the field arithmetic algorithms to the processor architecture to achieve acceptable timings for a scalar point multiplication. Running at 1 MHz we can perform a 128-bit random point multiplication in 3.4 sec using projective coordinates (this time includes transforming back to affine coordinates) using the binary method for exponentiation. Notice that these timings can be further improved by using addition-subtraction methods. Finally, it is possible to dramatically reduce the time required for a point multiplication if the point is known ahead of time, like in the ECDSA signature generation operation. This can be accomplished by using precomputation methods. However, these methods require additional memory which in embedded systems is not always freely available.

References

1. ANSI X9.62-1-xxxx. Public Key Cryptography for the Financial Services Industry: the Ellip tic Curve Digital Signature Algorithm (ECDSA) (Revised). Technical report, American Bankers Association, October 1999.
2. ANSI X9.62-1999. The Elliptic Curve Digital Signature Algorithm. Technical report, ANSI, 1999.

3. D. V. Bailey and C. Paar. Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume LNCS 1462, pages 472–485, Berlin, Germany, 1998. Springer-Verlag.
4. D. V. Bailey and C. Paar. Inversion in Optimal Extension Fields. In A. Odlyzko, G. Walsh, and H. Williams, editors, *Conference on The Mathematics of Public Key Cryptography*, The Fields Institute for Research in the Mathematical Sciences, Toronto, Canada, June 1999.
5. P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In A. M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume LNCS 263, pages 311–323, Berlin, Germany, August 1986. Springer-Verlag.
6. G. Borriello and R. Want. Embedded computation meets the world wide web. *Communications of the ACM*, 43(5):59–66, May 2000.
7. Ç. K. Koç, T. Acar, and B. Kaliski. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, pages 26–33, June 1996.
8. D.V. Chudnovsky and G.V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7:385–434, 1986.
9. Jae Wook Chung, Sang Gyoo Sim, and Pil Joong Lee. Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, pages 57–70, Berlin, 2000. Springer-Verlag.
10. Henry Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology — ASIACRYPT'98*, volume LNCS 1514, pages 51–65, Berlin, 1998. Springer-Verlag.
11. S. R. Dussé and B. S. Kaliski. A Cryptographic Library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, volume LNCS 473, pages 230–244, Berlin, Germany, May 1990. Springer-Verlag.
12. D. Estrin, R. Govindan, and J. Heidemann. Embedding the Internet. *Communications of the ACM*, 43(5):39–41, May 2000.
13. D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27:129–146, 1998.
14. J. Guajardo and C. Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. In B. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume LNCS 1294, pages 342–356, Berlin, Germany, August 1997. Springer-Verlag.
15. R. Harley, D. Doligez, D. de Rauglaudre, and X. Leroy. http://cristal.inria.fr/%7Eharley/ecdl7/.
16. Toshio Hasegawa, Junko Nakajima, and Mitsuru Matsui. A Practical Implementation of Elliptic Curve Cryptosystems over $GF(p)$ on a 16-bit Microcomputer. In Hideki Imai and Yuliang Zheng, editors, *First International Workshop on Practice and Theory in Public Key Cryptography — PKC'98*, volume LNCS 1431, pages 182–194, Berlin, 1998. Springer-Verlag.
17. K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara. Fast Implemenation of Public-Key Cryptography on a DSP TMS320C6201. In Çetin K. Koç and Christof Paar, editors, *Proceedings of the First Workshop on Cryptographic Hardware and Embedded Systems — CHES'99*, volume LNCS 1717, pages 61–72, Berlin, Germany, August 1999. Springer-Verlag.
18. T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.

19. D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, USA, 2nd edition, 1981.
20. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.
21. Arjen Lenstra and Eric Verheul. Selecting cryptographic key sizes. In Hideki Imai and Yuliang Zheng, editors, *Third International Workshop on Practice and Theory in Public Key Cryptography — PKC 2000*, volume LNCS 1751, Berlin, 2000. Springer-Verlag.
22. Chae Hoon Lim and Hyo Sun Hwang. Fast Implementation of Elliptic Curve Arithmetic in $GF(p^n)$. In Hideki Imai and Yuliang Zheng, editors, *Third International Workshop on Practice and Theory in Public Key Cryptography — PKC 2000*, volume LNCS 1751, pages 405–421, Berlin, 2000. Springer-Verlag.
23. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, Boston, Massachusetts, USA, 1993.
24. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, Florida, USA, 1997.
25. V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO '85*, volume LNCS 218, pages 417–426, Berlin, Germany, 1986. Springer-Verlag.
26. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
27. U.S. Department of Commerce/National Institute of Standard and Technol ogy. *Digital Signature Standard (DSS)*, January 27 2000.
28. National Institute of Standard and Technology. Recommended elliptic curves for federal government use. available at http://csrc.nist.gov/encryption, May 1999.
29. *IEEE P1363 Standard Specifications for Public Key Cryptography*, November 1999. Last Preliminary Draft.
30. R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO '95*, volume LNCS 963, pages 43–56, Berlin, Germany, 1995. Springer-Verlag.
31. Standards for Efficient Cryptography Group. SEC2: Recommended Elliptic Curve Domain Parameters. Working draft, version 0.7, September 2000.
32. Texas Instruments, Inc., Dallas, Texas 75265 USA. *MSP430C33x,MSP430P337A Mixed Signal Microcontrollers*, October 1999 (Revised June 2000).
33. Texas Instruments, Inc., Dallas, Texas 75265 USA. *MSP430x3xx Family – User's Guide*, July 2000.
34. A. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *IFIP CARDIS 2000, Fourth Smart Card Research and Advanced Application Conference*, Bristol, UK, September 20–22 2000. Kluwer.