ELLIPTIC CURVE CRYPTOGRAPHY ON SMART CARDS WITHOUT COPROCESSORS

Adam D. Woodbury
Electrical and Computer Engineering Department
adw@ece.wpi.edu

Daniel V. Bailey
Computer Science Department
bailey@cs.wpi.edu

Christof Paar
Electrical and Computer Engineering Department, Computer Science Department
christof@ece.wpi.edu

Worcester Polytechnic Institute
Worcester, MA 01609 USA

The Fourth Smart Card Research and Advanced Applications (CARDIS 2000) Conference, September 20-22, 2000, Bristol, UK.

Abstract This contribution describes how an elliptic curve cryptosystem can be implemented on very low cost microprocessors with reasonable performance. We focus in this paper on the Intel 8051 family of microcontrollers popular in smart cards and other cost-sensitive devices. The implementation is based on the use of the finite field $GF((2^8 - 17)^{17})$ which is particularly suited for low end 8-bit processors. Two advantages of our method are that subfield modular reduction can be performed infrequently, and that an adaption of Itoh and Tsujii's inversion algorithm is used for the group operation. We show that an elliptic curve scalar multiplication with a fixed point, which is the core operation for a signature generation, can be performed in a group of order approximately 2^{134} in less than 2 seconds. Unlike other implementations, we do not make use of curves defined over a subfield such as Koblitz curves.

Keywords: finite fields, fast arithmetic, Optimal Extension Fields, modular reduction, elliptic curves, implementation, smart cards, Intel 8051

1. INTRODUCTION AND MOTIVATION

A typical large-scale smart card application such as retail banking can entail the manufacture, personalization, issuance, and support of millions of smart cards. Due to the grand scale involved, the success of such an application is inherently linked to careful cost management of each of these areas. However, budgetary constraints must be weighed against the basic requirements for smart card security. The security services offered by a smart card often include both data encryption and public-key operations. Creation of a digital signature is often the most computationally intensive operation demanded of a smart card.

Smart cards often use 8-bit microcontrollers derived from 1970s families such as the Intel 8051 [25] and the Motorola 6805. The use of public-key algorithms such as RSA or DSA, which are based on modular arithmetic with very long operands, on such a processor predictably results in unacceptably long processing delays. To address this problem, many smart card microcontroller manufacturers include additional on-chip hardware to accelerate long-number arithmetic operations. However, in cost-sensitive applications it can be attractive to execute public-key operations on smart cards without coprocessors.

The challenge addressed in this contribution is to implement a public-key digital signature algorithm which does not introduce performance problems or require additional hardware beyond an 8-bit microcontroller. To address this problem, we turn to the computational savings made available by elliptic curve cryptosystems. An elliptic curve cryptosystem relies on the assumed hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP) for its security. An instance of the ECDLP is posed for an elliptic curve defined over a finite field $GF(p^m)$ for p a prime and m a positive integer. The rule to perform the elliptic curve group operation can be expressed in terms of arithmetic operations in the finite field; thus the speed of the field arithmetic determines the speed of the cryptosystem.

In this paper, we first compare the finite field arithmetic performance offered by three different types of finite field which have been proposed for elliptic curve cryptosystems (ECCs): binary fields $GF(2^n)$, even composite fields $GF((2^n)^m)$, and finally Optimal Extension Fields (OEFs): $GF(p^m)$ for p a pseudo-Mersenne prime, m chosen so that an irreducible binomial exists over $GF(p)$. Our results show that core field arithmetic operations in $GF(2^n)$ lag behind the other two at a ratio of 5:1. The arithmetic offered by OEFs and composite fields is comparable in performance. However, the recent result of Gaudry, Hess, and Smart [10] has shown that the ECDLP can be easily solved when even composite fields are used. Thus, in the main part of this paper we present the results of applying OEFs to the construction of ECCs to calculate a

digital signature within a reasonable processing time with no need for hardware beyond an 8-bit microcontroller. The target processor is an 8051, derivatives of which are on many popular smart cards such as the Siemens 44C200 and Phillips 82C852.

2. PREVIOUS WORK

This section reviews some of the most relevant previous contributions. It has been long recognized that efficient finite field arithmetic is vital to achieve acceptable performance with ECCs. Before an attack was published rendering them unattractive, many implementors chose even-characteristic finite fields with composite extension degree.

A paper due to De Win et.al. [8] analyzes the use of fields $GF((2^n)^m)$, with a focus on $n = 16$, $m = 11$. This construction yields an extension field with 2^{176} elements. The subfield $GF(2^{16})$ has a Cayley table of sufficiently small size to fit in the memory of a workstation. Optimizations for multiplication and inversion in such composite fields of characteristic two are described in [11].

Schroeppel et.al. [24] report an implementation of an elliptic curve analogue of Diffie-Hellman key exchange over $GF(2^{155})$. The arithmetic is based on a polynomial basis representation of the field elements. Another paper by De Win et.al. [9] presents a detailed implementation of elliptic curve arithmetic on a desktop PC, using finite fields of the form $GF(p)$ and $GF(2^n)$, with a focus on its application to digital signature schemes. For ECCs over prime fields, their construction uses projective coordinates to eliminate the need for inversion, along with a balanced ternary representation of the multiplier. Claus Schnorr presents a digital signature algorithm based on the finite field discrete logarithm problem in [23]. The algorithm is especially suited for smart cards.

The work in [1, 2] introduces OEFs and provides performance statistics on high-end RISC workstations. A paper extending the work on OEFs appears in [16]. In this paper, sub-millisecond performance on high-end RISC workstations is reported. Further, the authors achieve an ECC performance of 1.95 msec on a 400 MHz Pentium II. A rump session presentation in [20] introduces an efficient algorithm for exponentiation in an OEF which leads to efficient implementation of cryptosystems based on the finite field discrete logarithm problem. Reference [3] introduces the Itoh-Tsujii inversion algorithm for OEFs which is used in this contribution.

In [21], Naccache and M'Raïhi provide an overview of smart cards with cryptographic capabilities, including a discussion of general implementation concerns on various types of smart cards. In [22] a zero-knowledge system on an 8-bit microprocessor without a coprocessor is presented.

In a white paper [6], Certicom Corp. provides performance data for an ECC defined over $GF(2^{163})$ on smart card CPUs without cryptographic coproces-

sors. Statistics on the performance of the finite field arithmetic operations are not included. In addition, no details are provided about the particular elliptic curve they chose as a basis for their implementation. When a Siemens SLE44C80S is used as the smart card microcontroller, digital signature performance of under 1.5 seconds is reported. An improved timing of 700 msec is reported for a Siemens SLE66C80S, a 16-bit microcontroller. These processors are variants of the Intel 8051 and hence these results are directly relevant to those achieved in this paper.

3. FINITE FIELD CHOICE

To implement an ECC, an implementor must select a finite field in which to perform arithmetic calculations. A finite field is identified with the notation $GF(p^m)$ for p a prime and m a positive integer. It is well known that there exists a finite field for all primes p and positive rational integers m. This field is isomorphic to $GF(p)[x]/(P(x))$, where $P(x) = x^m + \sum_{i=0}^{m-1} p_i\ x^i, p_i \in GF(p)$, is a monic irreducible polynomial of degree m over $GF(p)$. In the following, a residue class will be identified with the polynomial of least degree in this class.

Various finite fields admit the use of different algorithms for arithmetic. Unsurprisingly, the choices of p, m, and $P(x)$ can have a dramatic impact on the performance of the ECC. In particular, there are generic algorithms for arithmetic in an arbitrary finite field and there are specialized algorithms which provide better performance in finite fields of a particular form. In the following, we briefly describe field types proposed for ECC.

3.1. BINARY FIELDS

Implementors designing custom hardware for an ECC often choose $p = 2$ and $P(x)$ to be a trinomial or pentanomial. Such choices of irreducible polynomial lead to efficient methods for extension field modular reduction. We will refer to this type of field as a "binary field," in accordance with [13]. The elements of the subfield $GF(2)$ can be represented by the logical signals 0 and 1. In this way, it is both speed and area efficient to construct hardware circuits to perform the finite field arithmetic.

3.2. EVEN COMPOSITE FIELDS

In software, the choice of parameters varies considerably with the wide array of available microprocessors. Many authors have suggested the use of $p = 2$ and m a composite number. In this case, the field $GF(2^m)$ is isomorphic to $GF((2^s)^r)$, for $m = sr$ and we call this an "even composite field." Then multiplication and inversion in the subfield $GF(2^s)$ can be efficiently performed by table look-up if s is not too large. In turn, these operations in the extension

field $GF((2^s)^r)$ are calculated using arithmetic in the subfield. As in the binary field case, the irreducible polynomials for both the subfield and the extension field are chosen to have minimal weight. This approach can provide superior performance when compared to the case of binary fields. *However, a recent attack against ECCs over composite fields [10] makes them inappropriate for use in practice.*

3.3. OPTIMAL EXTENSION FIELDS

An alternative construction is to use OEFs [2], which choose p of the form $2^n \pm c$, for n, c arbitrary positive rational integers. In this case, one chooses p of appropriate size to use the multiply instructions available on the target microcontroller. In addition, m is chosen so that an irreducible binomial $P(x) = x^m - \omega$ exists.

3.4. ROUGH PERFORMANCE COMPARISON

To address our need for fast field arithmetic in an ECC implemented on a smart card, we compared these three options for finite field arithmetic on a standard Intel 8051 running at 12 MHz. Due to the 8051's internal clock division factor of 12, one internal clock cycle is equivalent to one microsecond. Thus, these timings may be interpreted as either internal clock cycles or microseconds. We implemented extension field multiplication for the three candidates in assembly. We chose a field order of about 2^{135} which provides moderate security as will be discussed in Section 3.5 below. Field multiplication is the time critical operation in most ECC realizations. We represented field elements with a polynomial basis and took advantage of the standard arithmetic algorithms available for each. Results are shown in Table 1.

Table 1 Extension field multiplication performance on an Intel 8051

Field	*appr. Field Order*	*# Cycles for Multiply*
$GF(2^{135})$	2^{135}	19,600
$GF((2^8)^{17})$	2^{136}	7,479
$GF((2^8-17)^{17})$	2^{134}	5,084

Thus we see that binary fields offer performance which lags far behind the other two options. Further, even composite fields have recently been shown to have cryptographic weaknesses [10]. Hence, we are lead to conclude that OEFs are the best choice for our application.

3.5. REMARK ON THE FINITE FIELD ORDER CHOSEN

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 [17]. 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 [12]. First, note that the field $GF((2^8 - 17)^{17})$ has an order of about 2^{134}. 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 134-bit field which does not use a subfield curve is by a factor of $\sqrt{108} \cdot \sqrt{2^{26}} \approx 2^{16}$ 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 72-bit block cipher. This implies that an attack would require about 65,000 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 $GF(239^{17})$ offers far more security than the 512-bit RSA system which has been popular for fielded smart card applications. In summary, we feel that our selection of field order provides medium-term security which is sufficient for many current smart card applications.

Of course, the discussion above assumes that there are no special attacks against ECC over OEFs. This assumption seems to be valid at the time of writing [10].

To generate good elliptic curves over OEFs there are two basic approaches. The first one is based on the use of a curve defined over $GF(p)$ using the method in [4, Section VI.4]. The second, more general, method uses Schoof's algorithm together with its improvements. The algebra package LiDIA v2.0.1 supports EC point counting over arbitrary fields.

4. ALGORITHMS FOR AN 8-BIT MICROCONTROLLER

When choosing an algorithm to implement on 8-bit processors, it is important that the parameter choices match the target platform. The Intel 8051 offers a multiply instruction which computes the product of two integers each less than $2^8 = 256$. Thus, we chose a prime $2^8 - 17 = 239$ as our field characteristic so that multiplication of elements in the prime subfield can use the ALU's multiplier. In addition, the nature of the OEF leads to an efficient reduction method. Field elements are represented as polynomials of degree up to 16,

with coefficients in the prime subfield $GF(239)$. As mentioned in Section 3.3, the polynomial is reduced modulo an irreducible polynomial, $P(x) = x^m - \omega$. In this implementation $P(x) = x^{17} - 2$.

The key performance advantage of OEFs is due to fast modular reduction in the subfield. Given a prime, $p = 2^n - c$, reduction is performed by dividing the number x into two n-bit words. The upper bits of x are "folded" into the lower ones, leading to a very efficient reduction. The basic reduction step which reduces a $2n$-bit value x to a result with $1.5n$ bits is given by representing $x = x_1 2^n + x_0$, where $x_0, x_1 < 2^n$. Thus a reduction is performed by:

$$x \equiv x_1 c + x_0 \bmod 2^n - c, \tag{1}$$

which takes one multiplication by c, one addition, and no divisions or inversions. As will be seen in Section 4.1, the reduction principle for OEFs is expanded for the implementation described here.

Furthermore, calculating a multiplicative inverse over the 8-bit subfield is easily implemented with table look-up. There is a relative cost in increased codesize, but the subfield inverse requires only two instructions. In contrast, a method such as the Extended Euclidian Algorithm would require a great deal more processing time. This operation is required for our optimized inversion algorithm, as described in Section 4.3.

For elliptic curves, extension field multiplication is the most important basic operation. The elliptic curve group operation requires 2 multiplications, 1 squaring, 1 inversion, and a number of additions that are relatively fast compared with the first three. In our case, squaring and inversion performance depends on the speed of multiplication. Therefore the speed of a single extension field multiplication defines the speed of the group operation in general.

Addition is carried out in the extension field by $m - 1$ component-wise additions modulo p. Subtraction is performed in a similar manner.

4.1. MULTIPLICATION

Extension field multiplication is implemented as polynomial multiplication with a reduction modulo the irreducible binomial $P(x) = x^{17} - 2$. This modular reduction is implemented in an analogous manner to the subfield modular reduction outlined above. First, we observe that $x^m \equiv \omega \bmod x^m - \omega$. This observation leads to the general expression for this reduction, given by

$$\begin{aligned} C(x) \quad \equiv \quad & c'_{m-1}x^{m-1} + [\omega c'_{2m-2} + c'_{m-2}]x^{m-2} + \cdots \\ & + [\omega c'_{m+1} + c'_1]x + [\omega c'_m + c'_0] \bmod x^m - \omega. \end{aligned} \tag{2}$$

Thus, product C of a multiplication $A \times B$ can be computed as shown in Algorithm 1.1.

Algorithm 1.1 Extension Field Multiplication

Require: $A(x) = \sum a_i x^i, B(x) = \sum b_i x^i \in GF(239^{17})/P(x)$, where $P(x) = x^m - \omega$; $a_i, b_i \in GF(239)$; $0 \leq i < 17$

Ensure: $C(x) = \sum c_i x^i = A(x)B(x)$, $c_i \in GF(239)$

First we calculate the intermediate values for c'_i, $i = 17, 18, \ldots, 32$.

$c'_{17} \leftarrow a_1 b_{16} + a_2 b_{15} + \ldots + a_{14} b_3 + a_{15} b_2 + a_{16} b_1$

$c'_{18} \leftarrow a_2 b_{16} + a_3 b_{15} + \ldots + a_{15} b_3 + a_{16} b_2$

$\ldots$

$c'_{31} \leftarrow a_{15} b_{16} + a_{16} b_{15}$

$c'_{32} \leftarrow a_{16} b_{16}$

Now calculate c_i, $i = 0, 1, \ldots, 16$.

$c_0 \leftarrow a_0 b_0 + \omega c'_{17} \bmod 239$

$c_1 \leftarrow a_0 b_1 + a_1 b_0 + \omega c'_{18} \bmod 239$

$\ldots$

$c_{15} \leftarrow a_0 b_{15} + a_1 b_{14} + \ldots + a_{14} b_1 + a_{15} b_0 + \omega c'_{32} \bmod 239$

$c_{16} \leftarrow a_0 b_{16} + a_1 b_{15} + \ldots + a_{14} b_2 + a_{15} b_1 + a_{16} b_0 \bmod 239$

As can be seen, extension field multiplication requires m^2 inner products $a_i b_j$, and $m-1$ multiplications by ω when the schoolbook method for polynomial multiplication is used. These m^2+m-1 subfield multiplications form the performance critical part of a field multiplication. In the earlier OEF work [1], [2], a subfield multiplication was performed as single-precision integer multiplication resulting in a double-precision product with a subsequent reduction modulo p. For OEFs with $p = 2^n \pm c$, $c > 1$, this approach requires 2 integer multiplications and several shifts and adds using Algorithm 14.47 in [19]. A key idea of this contribution is to deviate from this approach. We propose to perform only one reduction modulo p per coefficient c_i, $i = 0, 1, \ldots, 16$. This is achieved by allowing the residue class of the sum of integer products to be represented by an integer larger than p. The remaining task is to efficiently reduce a result which spreads over more than two words. Hence, we can reduce the number of reductions to m, while still requiring m^2+m-1 multiplications.

During the inner product calculations, we perform all required multiplications for a resulting coefficient, accumulate a multi-word integer, and then perform a reduction. The derivation of the maximum value for the multi-word integer c_i before reduction is shown in Table 2.

We now expand the basic OEF reduction shown in Equation (1) for multiple words. As the $\log_2(\text{ACC}_{max}) = 21$ bits, the number can be represented in the radix 2^8 with three digits. We observe $2^n \equiv c \pmod{2^n - c}$ and $2^{2n} \equiv c^2 \pmod{2^n - c}$. Thus the expanded reduction for operands of this size is performed by representing $x = x_2 2^{2n} + x_1 2^n + x_0$, where $x_0, x_1, x_2 < 2^n$.

Table 2 Inner product maximum value

1	one inner product multiplication with a maximum value of $(p-1)^2$
2	we accumulate 17 products, 16 of which are multiplied by $\omega = 2$
3	$\text{ACC}_{max} = 33(p-1)^2 = 1869252 = \text{1C85C4h} < 2^{21}$

Table 3 Intermediate reduction maxima

1	Using Equation (3), given that $0 \leq x \leq$ 1C85C4h
2	**max**$(x') =$ 1734h, when $x =$ 1BFFFFh.
3	Using Equation (4), given that $0 \leq x' \leq$ 1734h
4	**max**$(x'') =$ 275h, when $x' =$ 16FFh.

The first reduction is performed as

$$x' \equiv x_2 c^2 + x_1 c + x_0 \ (\mathrm{mod} 2^n - c), \tag{3}$$

noting that $c^2 = 289 \equiv 50 \bmod 239$. The reduction is repeated, now representing the previous result as $x' = x'_1 2^n + x'_0$, where $x'_0, x'_1 < 2^n$. The second reduction is performed as

$$x'' \equiv x'_1 c + x'_0 \bmod 2^n - c. \tag{4}$$

The maximum intermediate values through the reduction are shown in Table 3. Step 1 shows the maximum sum after inner product addition. While this value is the largest number that will be reduced, it is more important to find the maximum value that can result from the reduction. This case can be found by maximizing x_1 and x_0 at the cost of reducing x_2 by one. Looking at Table 3 again, this value is shown in step 2, as is the resulting reduced value. The process is repeated again in steps 3 and 4, giving us the maximum reduced value after two reductions.

Note that through two reductions, we reduced a 21-bit input to 13 bits, and finally to 10 bits. At this point in the reduction, we could perform the same reduction again, but it would only provide a slight improvement. Adding $x''_1 c + x''_0$ would result in a 9-bit number. Therefore it is much more efficient to handle each possible case. Most important is to eliminate the two high bits, and then to

ensure the resulting 8-bit number is the least positive representative of its residue class. The entire multiplication and reduction is shown in Algorithm 1.2.

To perform the three-word reduction requires three 8-bit multiplications and then several comparative steps. After the first two multiplications, the inner product sum has been reduced to a 13-bit number. If we were to reduce each inner product individually, every step starting at line 13 in Algorithm 1.2 would be required. Ignoring the trailing logic, which would add quite a bit of time itself, this would require $m = 17$ multiplications as opposed to the three needed in Algorithm 1.2. By allowing the inner products to accumulate and performing a single reduction we have saved 14 multiplications, plus additional time in trailing logic, per coefficient calculation. Recall that we require 17 coefficient calculations per extension field multiplication.

4.2. SQUARING

Extension field squaring is similar to multiplication, except that the two inputs are equal. By modifying the standard multiplication routine, we are able to take advantage of identical inner product terms. For example, $c_2 = a_0b_2 + a_1b_1 + a_2b_0 + \omega c_{19}$, can be simplified to $c_2 = 2a_0a_2 + {a_1}^2 + \omega c_{19}$. Further gain is accomplished by doubling only one coefficient, reducing it, and storing the new value. This approach saves us from recalculating the doubled coefficient when it is needed again. A side benefit of all the effort is that the maximum inner product value is slightly lower. The exact inner product maximum is 177F8h, but this makes little difference to the reduction algorithm. After two general OEF reductions, the maximum is reduced to 242h. As this is still a 10-bit number, the next reduction steps would be identical to their multiplication counterparts, and therefore the same reduction code is used.

4.3. INVERSION

Inversion in the OEF is performed via a modification of the Itoh-Tsujii algorithm [14] as shown in [3], which reduces the problem of extension field inversion to subfield inversion. The algorithm computes an inverse in $GF(p^{17})$ as $A^{-1} = (A^r)^{-1}A^{r-1}$ where $r = (p^{17} - 1)/(p - 1) = 11\ldots10_p$. Algorithm 1.3 shows the details of this method. A key point is that $A^r \in GF(p)$ and is therefore an 8-bit value. Therefore the step shown in line 10 is only a partial extension field multiplication, as all coefficients of A^r other than b_0 are zero. Inversion of A^r in the 8-bit subfield is performed via a table look-up.

The most costly operation is the computation of A^r. Because the exponent is fixed, an addition chain can be derived to perform the exponentiation. For $m = 17$, the addition chain requires 4 multiplications and 5 exponentiations to a p^i-th power. The element is then inverted in the subfield, and then multiplied back in. This operation results in the field inverse.

Algorithm 1.2 Extension Field Multiplication with Subfield Reduction

Require: $A(x) = \sum a_i x^i, B(x) = \sum b_i x^i \in GF(239^{17})/P(x)$, where $P(x) = x^m - \omega;\ a_i, b_i \in GF(239);\ 0 \leq i < 17$
Ensure: $C(x) = \sum c_i x^i = A(x)B(x),\ c_i \in GF(239)$

Define $z[w]$ to mean the w-th 8-bit word of z
$c_i \leftarrow 0$
if $i \neq 16$ **then**
  for $j \leftarrow m-1$ downto $i+1$ **do**
    $c_i \leftarrow c_i + a_{i+m-j} b_j$
  end for
  $c_i \leftarrow 2c_i$ – multiply by $\omega = 2$
end if
for $j \leftarrow i$ downto 0 **do**
  $c_i \leftarrow c_i + a_{i-j} b_j$
end for
$c_i \leftarrow c_i[2] * 50 + c_i[1] * 17 + c_i[0]$ – begin reduction, Equation (3)
$t \leftarrow c_i[1] * 17$ – begin Equation (4)
if $t \geq 256$ **then**
  $t \leftarrow t[0] + 17$
end if
$c_i \leftarrow c_i[0] + t$ – end Equation (4)
if $c_i \geq 256$ **then**
  $c_i \leftarrow c_i[0] + 17$
  if $c_i \geq 256$ **then**
    $c_i \leftarrow c_i[0] + 17$
    terminate
  end if
end if
$c_i \leftarrow c_i - 239$
if $c_i \leq 0$ **then**
  $c_i \leftarrow c_i + 239$
end if

Table 4 Frobenius constants $B(x) = A(x)^{p^i}$

	Exponent			
Coefficient	p	p^2	p^4	p^8
a_0	1	1	1	1
a_1	132	216	51	211
a_2	216	51	211	67
a_3	71	22	6	36
a_4	51	211	67	187
a_5	40	166	71	22
a_6	22	6	36	101
a_7	36	101	163	40
a_8	211	67	187	75
a_9	128	132	216	51
a_{10}	166	71	22	6
a_{11}	163	40	166	71
a_{12}	6	36	101	163
a_{13}	75	128	132	216
a_{14}	101	163	40	166
a_{15}	187	75	128	132
a_{16}	67	187	75	128

The Frobenius map raises a field element to the p-th power. In practice, this automorphism is evaluated in an OEF by multiplying each coefficient of the element's polynomial representation by a "Frobenius constant," determined by the field and its irreducible binomial. A list of the constants used is shown in Table 4. To raise a given field element to the p^i-th power, each a_j, $j = 0, 1, \ldots, 16$, coefficient are multiplied by the corresponding constant in the subfield $GF(239)$.

Thus we have efficient methods for both the exponentiation and subfield inversion required in Algorithm 1.3. Our results in Section 6 show the ratio of multiplication time to inversion time is 1:4.8. This ratio indicates that an affine representation of the curve points offers better performance than the corresponding projective-space approach, which eliminates the need for an inversion in every group operation at the expense of many more multiplications.

4.4. GROUP OPERATION

The operation in the Abelian group of points on an elliptic curve is called "point addition." This operation adds two curve points, and results in another point on the curve. Using an ECC for signatures involves the repeated ap-

Algorithm 1.3 Inversion Algorithm in $GF((2^8-17)^{17})$

Require: $A \in GF(p^{17})$
Ensure: $B \equiv A^{-1} \bmod P(x)$
1: $B_0 \leftarrow A^p = A^{(10)_p}$
2: $B_1 \leftarrow B_0 A = A^{(11)_p}$
3: $B_2 \leftarrow (B_1)^{p^2} = A^{(1100)_p}$
4: $B_3 \leftarrow B_2 B_1 = A^{(1111)_p}$
5: $B_4 \leftarrow (B_3)^{p^4} = A^{(11110000)_p}$
6: $B_5 \leftarrow B_4 B_3 = A^{(11111111)_p}$
7: $B_6 \leftarrow (B_5)^{p^8} = A^{(1111111100000000)_p}$
8: $B_7 \leftarrow B_6 B_5 = A^{(1111111111111111)_p}$
9: $B_8 \leftarrow (B_7)^p = A^{(1111111111111110)_p}$
10: $b \leftarrow B_8 A = A^{r-1} A = A^r$
11: $b \leftarrow b^{-1} = (A^r)^{-1}$
12: $B \leftarrow b B_8 = (A^r)^{-1} A^{r-1} = A^{-1}$

plication of the group law. The group law using affine coordinates is shown below [18].

If $P = (x_1, y_1) \in GF(p^m)$, then $-P = (x_1, -y_1)$. If $Q = (x_2, y_2) \in GF(p^m), Q \neq -P$, then $P + Q = (x_3, y_3)$, where

$$x_3 = \lambda^2 - x_1 - x_2, \tag{5}$$

$$y_3 = \lambda(x_1 - x_3) - y_1, \tag{6}$$

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, & \text{if } P \neq Q, \\ \frac{3x_1^2 + a}{2y_1}, & \text{if } P = Q. \end{cases} \tag{7}$$

The λ term is calculated depending on the relationship of P and Q. If they are equal, then a point doubling is performed, using the second equation. Note that λ is undefined if the points are additive inverses, or if either point is zero. These conditions must be examined before the group operation is performed.

4.5. POINT MULTIPLICATION

The operation required in an ECC is point multiplication, denoted by kP, where k is an integer and P is a point on the curve. For large k, computing kP is a costly endeavor. However, well-studied techniques used for ordinary integer exponentiation can be advantageously applied. The most basic of these algorithms is the binary-double-and-add algorithm [15]. It has a complexity of $\log_2(k) + H(k)$ group operations, where H is the Hamming weight of

the multiplier k. On average, then, we can expect this algorithm to require $1.5\log_2(k)$ group operations. Using more advanced methods, such as signed digit, k-ary or sliding window, the complexity may be reduced to approximately $1.2\log_2(k)$ group operations on average [19].

The situation is much better in certain applications, however. The most common public-key operation for a smart card is to provide a digital signature. The ECDSA algorithm [13] involves the multiplication of a public fixed curve point by the user generated private key as the core operation. Because the curve point is known ahead of time, precomputations may be performed to expedite the signing process. Using a method devised by de Rooij in [7], we are able to reduce the number of group operations necessary by a factor of four over the binary-double-and-add algorithm. The de Rooij algorithm is a variant of that devised by Brickell, Gordon, McCurley, and Wilson [5], but requires far fewer precomputations.

Algorithm 1.4 EC Fixed Point Multiplication using Precomputation and Vector Addition Chains

Require: $\{b^0A, b^1A, \ldots, b^tA\}$, $A \in E(GF(p^m))$, and $s = \sum_{i=0}^{t} s_i b^i$
Ensure: $C = sA$, $C \in E(GF(p^m))$
1: Define $M \in [0,t]$ such that $z_M \geq z_i$ for all $0 \leq i \leq t$
2: Define $N \in [0,t], N \neq M$ such that $z_N \geq z_i$ for all $0 \leq i \leq t, i \neq M$
3: **for** $i \leftarrow 0$ to t **do**
4: $\quad A_i \leftarrow b^iA$
5: $\quad z_i \leftarrow s_i$
6: **end for**
7: Determine M and N for $\{z_0, z_1, \ldots, z_t\}$
8: **while** $z_N \geq 0$ **do**
9: $\quad q \leftarrow \lfloor z_M / z_N \rfloor$
10: $\quad A_N \leftarrow qA_M + A_N$ – general point multiplication
11: $\quad z_M \leftarrow z_M$ **mod** z_N
12: $\quad$ Determine M and N for $\{z_0, z_1, \ldots, z_t\}$
13: **end while**
14: $C \leftarrow z_M A_M$

A modified form of de Rooij is shown in Algoritm 1.4. Note that the step shown in line 10 requires general point multiplication of A_M by q, where $0 \leq q < b$. This is accomplished using the binary-double-and-add algorithm. In [7], the author remarks that during execution, q is rarely greater than 1.

The choice of t and b are very important to the operation of this algorithm. They are defined such that $b^{t+1} \geq \#E(GF(p^m))$. The algorithm must be able to handle a multiplier, s, not exceeding the order of the elliptic curve. The number of point precomputations and temporary storage locations is determined

by $t + 1$, while b represents the maximum size of the exponent words. Thus we need to find a compromise between the two parameters.

Two obvious choices for an 8-bit architecture are $b = 2^{16}$ and $b = 2^8$, since dividing the exponent into radix b words is essentially free as they align with the memory structure. This results in a precomputation count of 9 and 18 points, respectively. The tradeoff here is the cost of memory access vs. arithmetic speeds. As we double the number of precomputed points, the algorithm operates only marginally faster, as shown in [7], but the arithmetic operations are easier to perform on the 8-bit microcontroller. The problem is that the time to access such large quantities of data, 34 bytes per precomputed point and storage location in external RAM (XRAM), adds up. Note that even though the XRAM may be physically internal to the microcontroller, it is outside the natural address space, and thus a time delay is incurred for access.

For $b = 2^{16}$, we must perform 16-bit multiplication and modular reduction, but only need to store 9 precomputed points and 9 temporary points. For $b = 2^8$, however, we must now store 18 precomputed points and 18 temporary points, but now only have to perform 8-bit multiplication and modular reduction. Implementation results show that the speed gain from doubling the precomputations and the faster 8-bit arithmetic slightly outweighs the cost of the increase in data access, as shown in Section 6, assuming a microcontroller with enough XRAM is available.

5. IMPLEMENTATION DETAILS

Implementing ECCs on the 8051 is a challenging task. The processor has only 256 bytes of internal RAM available, and only the lower 128 bytes are directly addressable. The upper 128 bytes must be referenced through the use of the two pointer registers: R0 and R1. Accessing this upper half takes more time per operation and incurs more overhead in manipulating the pointers. To make matters worse, the lower half of the internal RAM must be shared with the system registers and the stack, thus leaving fewer memory locations free. XRAM may be utilized, but there is essentially only a single pointer for these operations, which are at typically at least three times slower than their internal counterparts.

This configuration makes the 8051 a tight fit for an ECC. Each curve point in our group occupies 34 bytes of RAM, 17 bytes each for the X and Y coordinates. To make the system as fast as possible, the most intensive field operations, such as multiplication, squaring, and inversion, operate on fixed memory addresses in the faster, lower half of RAM. During a group operation, the upper 128 bytes are divided into three sections for the two input and one output curve points, while the available lower half of RAM is used as a working area for the field arithmetic algorithms. A total of four 17-byte coordinate locations are used,

Table 5 Internal RAM memory allocation

Address	*Function*
00–07h	Registers
08–14h	de Rooij Algorithm Variables
15–35h	Call Stack (variable size)
36–3Bh	Pointers to Curve Points in Upper RAM
3C–7Fh	Temporary Field Element Storage
80–E5h	Temporary Curve Point Storage
E6–FFh	Unused

starting from address 3Ch to 7Fh, the top of lower RAM. This is illustrated in Table 5.

Finally, six bytes, located from 36h to 3Bh, are used to keep track of the curve points, storing the locations of each curve point in the upper RAM. Using these pointers, we can optimize algorithms that must repeatedly call the group operation, often using the output of the previous step as an input to the next step. Instead of copying a resulting curve point from the output location to an input location, which involves using pointers to move 34 bytes around in upper RAM, we can simply change the pointer values and effectively reverse the inputs and outputs of the group operation.

The arithmetic components are all implemented in handwritten, loop-unrolled assembly. This results in large, but fast and efficient program code, as shown in Table 7. Note that the execution times are nearly identical to the code size, an indication of their linear nature. Each arithmetic component is written with a clearly defined interface, making them completely modular. Thus, a single copy of each component exists in the final program, as each routine is called repeatedly.

Extension field inversion is constructed using a number of calls to the other arithmetic routines. The group operation is similarly constructed, albeit with some extra code for point equality and inverse testing. The binary-double-and-add and de Rooij algorithms were implemented in C, making calls to the group operation assembly code when needed. Looping structures were used in both programs as the overhead incurred is not as significant as it would be inside the group operation and field arithmetic routines. The final size and architecture requirements for the programs are shown in Table 6.

6. RESULTS

Our target microcontroller is the Siemens SLE44C24S, an 8051 derivative with 26 kilobytes of ROM, 2 kilobytes of EEPROM, and 512 bytes of XRAM.

Table 6 Program size and architecture requirements

Type	Size (bytes)	Function
Code	13k	Program Storage
Internal RAM	183	Finite Field Arithmetic
External RAM	306	Temporary Points
	34	Integer Multiplicand
Fixed Storage	306	Procomputed Points

Table 7 Finite field arithmetic performance on a 12 MHz 8051

Description	Operation	Time [a] (μsec)	Code Size (bytes)
Multiplication	$C(x) = A(x)B(x)$	5084	5110
Squaring	$C(x) = A^2(x)$	3138	3259
Addition	$C(x) = A(x) + B(x)$	266	360
Subtraction	$C(x) = A(x) - B(x)$	230	256
Inversion	$C(x) = A^{-1}(x)$	24489	[b]
Scalar Mult.	$C(x) = sA(x)$	642	666
Scalar Mult. by 2	$C(x) = 2A(x)$	180	257
Scalar Mult. by 3	$C(x) = 3A(x)$	394	412
Frobenius Map	$C(x) = A^{p^i}(x)$	625	886
Partial Multiplication	c_0 of $A(x)B(x)$	303	305
Subfield Inverse	$c = a^{-1}$	4	236

[a] Time calculated averaging over at least 5,000 executions with random inputs.
[b] Inversion is a collection of calls to the other routines and has negligible size itself.

This XRAM is in addition to the internal 256 bytes of RAM, and its use incurs a much greater delay. However, this extra memory is crucial to the operation of the de Rooij algorithm which requires the manipulation of several precomputed curve points.

The Keil PK51 tools were used to assemble, debug and time the algorithms, since we did not have access to a simulator for the Siemens smart card microcontrollers. Thus, to perform timing analysis a generic Intel 8051 was used, running at 12 MHz. Given the optimized architecture of the Siemens controller, an SLE44C24S running at 5 MHz is roughly speed equivalent to a 12 MHz Intel 8051.

Using each of the arithmetic routines listed in Table 7, the elliptic curve group operation takes 39.558 msec per addition and 43.025 msec per doubling on average.

Table 8 Elliptic curve performance on a 12 MHz 8051

Operation	Method	Time (msec)
Point Addition		39.558
Point Double		43.025
Point Multiplication	Binary Method	8370
Point Multiplication	de Rooij w/9 precomp.	1950
Point Multiplication	de Rooij w/18 precomp.	1830

Using random exponents, we achieve a speed of 8.37 seconds for point multiplication using binary-double-and-add. This is exactly what would be predicted given the speed of point addition and doubling. If we fix the curve point and use the de Rooij algorithm discussed in Section 4.5, we achieve speeds of 1.95 seconds and 1.83 seconds, for 9 and 18 precomputations respectively. This is a speed up factor of well over 4:1 when compared to general point multiplication. Unfortunately, our target microcontroller, the SLE44C24S, only has 512 bytes of XRAM where we manipulate our precomputed points. Since we require 34 bytes per precomputed point, 18 temporary points will not fit in the XRAM, limiting us to 9 temporary points on this microcontroller. These results are summarized in Table 8.

7. CONCLUSIONS AND OUTLOOK

We demonstrated that a scalar multiplication of a fixed point of an EC can be performed in under 2 seconds on an 8051 microcontroller. This is the core operation for signature generation in the ECDSA scheme. Although the performance and security threshold may not allow the use of our implementation in all smart card applications, we believe that there are scenarios where these parameters offer an attractive alternative to more costly smart cards with coprocessors, especially if public-key capabilities are added to existing systems.

We also believe that our implementation can be further improved. In practice, a smart card with an 8051-derived microcontroller that can be clocked faster than the 5 MHz assumed in Section 6 can obviously also easily yield point multiplication times which are below one second. In addition, 16-bit smart card microcontrollers such as the Siemens SLE66C80S would allow for a larger subfield and smaller extension degree, thus reaping immense benefits in field arithmetic algorithms. Further, the use of an elliptic curve defined over the prime subfield, as suggested in [16], could also provide a speedup. Each of these potential improvements provides further possibilities to apply the fast field arithmetic provided by an OEF to construct ECCs on smart card microcontrollers without additional coprocessors.

8. ACKNOWLEDGEMENTS

The authors would like to thank Jorge Guajardo and Pedro Soria-Rodriguez for their contribution of the even composite field multiplication implementation.

References

[1] Daniel V. Bailey. Optimal Extension Fields. Major Qualifying Project (Senior Thesis), 1998. Computer Science Department, Worcester Polytechnic Institute, Worcester, MA, USA.

[2] Daniel V. Bailey and Christof Paar. Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms. In *Advances in Cryptology – CRYPTO '98*. Springer-Verlag Lecture Notes in Computer Science, 1998.

[3] Daniel V. Bailey and Christof Paar. Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography. *Journal of Cryptology*, to appear.

[4] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[5] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In *Advances in Cryptography — EUROCRYPT '92*, pages 200–207. Springer-Verlag, 1993.

[6] Certicom Corp. The Elliptic Curve Cryptosystem for Smart Cards. online white paper, http://www.certicom.ca/ecc/wecc4.htm, 1998.

[7] Peter de Rooij. Efficient exponentiation using precomputation and vector addition chains. In *Advances in Cryptography — EUROCRYPT '98*, pages 389–399. Springer-Verlag, 1998.

[8] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. In *Asiacrypt '96*. Springer-Verlag Lecture Notes in Computer Science, 1996.

[9] E. De Win, S. Mister, B. Preneel, and M. Wiener. On the Performance of Signature Schemes Based on Elliptic Curves. In *Algorithmic Number Theory: Third International Symposium*, pages 252–266, Berlin, 1998. Springer-Verlag Lecture Notes in Computer Science.

[10] P. Gaudry, F. Hess, and N. P. Smart. Constructive and Destructive Facets of Weil Descent on Elliptic Curves. technical report HPL 2000-10, http://www.hpl.hp.com/techreports/2000/HPL-2000-10.html, 2000.

[11] Jorge Guajardo and Christof Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. In *Advances in Cryptology — Crypto '97*, pages 342–356. Springer-Verlag Lecture Notes in Computer Science, August 1997.

[12] R. Harley, D. Doligez, D. de Rauglaudre, and X. Leroy. http://cristal.inria.fr/%7Eharley/ecdl7/.

[13] IEEE. Standard Specifications for Public Key Cryptography. Draft, IEEE P1363 Standard, 1999. working document.

[14] 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.

[15] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[16] Tetsutaro Kobayashi, Hikaru Morita, Kunio Kobayashi, and Fumitaka Hoshino. Fast Elliptic Curve Algorithm Combining Frobenius Map and Table Reference to Adapt to Higher Characteristic. In *Advances in Cryptography — EUROCRYPT '99*. Springer-Verlag Lecture Notes in Computer Science, 1999.

[17] Arjen Lenstra and Eric Verheul. Selecting cryptographic key sizes. In *Public Key Cryptography — PKC 2000*. Springer-Verlag Lecture Notes in Computer Science, 2000.

[18] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[20] P. Mihăilescu. Optimal Galois field bases which are not normal. Fast Software Encryption rump session, 1997.

[21] D. Naccache and D. M'Raïhi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, 1996.

[22] D. Naccache, D. M'Raïhi, W. Wolfowicz, and A. di Porto. Are crypto-accelerators really inevitable? In *Advances in Cryptography — EUROCRYPT '95*, pages 404–409. Springer-Verlag Lecture Notes in Computer Science, 1995.

[23] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[24] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. *Advances in Cryptology — CRYPTO '95*, pages 43–56, 1995.

[25] Sencer Yeralan and Ashutosh Ahluwalia. *Programming and Interfacing the 8051 Microcontroller*. Addison-Wesley Publishing Company, 1995.