Generic GF(2^m) Arithmetic in Software and its Application to ECC

André Weimerskirch[1⋆], Douglas Stebila[2⋆], and Sheueling Chang Shantz[3]

[1] Communication Security Group, Ruhr-Universität Bochum, Germany
weika@crypto.rub.de
[2] Department of Combinatorics & Optimization, University of Waterloo, Ontario, Canada
dstebila@uwaterloo.ca
[3] Sun Microsystems Laboratories, Mountain View, California, USA
sheueling.chang@sun.com

The Eighth Australasian Conference on Information Security and Privacy (ACISP 2003), 9-11 July 2003, Wollongong, Australia

Abstract. This work discusses generic arithmetic for arbitrary binary fields in the context of elliptic curve cryptography (ECC). ECC is an attractive public-key cryptosystem recently endorsed by the US government for mobile/wireless environments which are limited in terms of their CPU, power, and network connectivity. Its efficiency enables constrained, mobile devices to establish secure end-to-end connections. Hence the server side has to be enabled to perform ECC operations for a vast number of mobile devices that use variable parameters in an efficient way to reduce cost. We present algorithms that are especially suited to high-performance devices like large-scaled server computers. We show how to perform an efficient field multiplication for operands of arbitrary size, and how to achieve efficient field reduction for dense polynomials. We also give running times of our implementation for both general elliptic curves and Koblitz curves on various platforms, and analyze the results. Our new algorithms are the fastest algorithms for arbitrary binary fields in literature.

Keywords: Binary Fields, Dense Field Polynomials, Field Multiplication, Field Reduction, Elliptic Curves, Koblitz Curves

1 Introduction

In recent years Elliptic Curve Cryptography (ECC) has received increased acceptance. ECC has been included in standards from bodies such as ANSI, IEEE, ISO, and NIST. Further evidence of widespread acceptance are the inclusion

⋆ The research was done at and sponsored by Sun Microsystems Laboratories, Mountain View, CA, USA.

of ECC in IPsec, TLS, and OpenSSL [17]. Compared to traditional cryptosystems like RSA, ECC offers smaller key sizes and more efficient arithmetic which results in faster computations, lower power consumption, as well as memory and bandwidth savings. This is especially useful for mobile, constrained devices. Hence ECC enables wireless mobile devices to perform secure communications efficiently and establish secure end-to-end connections.

In this paper we focus on binary field arithmetic and its application to ECC. Many software implementations have been reported in recent years. These works range from implementations in very constrained environments [1][23] to broad surveys [6]. However, all of these implementations are done for specific fields and cannot handle arbitrary fields and curves. While there is value in constructing a highly specialized implementation for constrained devices such as PDAs and smart cards, there is a need to handle arbitrary curves on the server side efficiently. As ECC becomes more important and more widely used, we foresee the following scenario. Commercial entities such as financial services or online stores will carry out transactions with users. The servers are required to perform cryptographic operations such as signature generation and verification, and key exchange. Most of these operations will employ standardized settings such as the NIST recommended elliptic curves [16]. However, there may be users that generate curves themselves, or Certificate Authorities that issue certificates over non-standardized curves that promise a performance gain. For example, they might want to use a curve defined over $GF(2^{155})$ as described in [18], a curve over $GF(2^{178})$ as presented in [19], or curves suited to special field element representations as in [2]. There may also be standards which are rarely used and thus do not justify the implementation of a special case, and future standards may introduce new curves. Still, a server must be able to handle all requests efficiently to reduce computing time and hence cost.

Our approach and motivation is based on this scenario. We present a binary field implementation for arbitrary fields $GF(2^m)$ in the context of ECC which works for CPUs with different word sizes. Our implementation is a complete software package which is part of the ECC support for binary fields of OpenSSL [17][1]. We present algorithms that are especially suited to high-performance devices like large-scaled server computers. We show how to perform an efficient field multiplication for operands of arbitrary size, and how to achieve efficient field reduction for dense polynomials. We give performance numbers that show the efficiency of our implementation, and analyze the results. We implemented both general curves and Koblitz curves. The results show that our algorithms are the fastest methods for arbitrary binary field arithmetic in existing literature.

The remainder of this paper is organized as follows. Related work is presented in the next section. Section 3 presents the algorithms that we used for the binary field arithmetic and the ECC operations, and Section 4 presents results and an analysis of our implementation.

[1] Note that this paper describes a highly optimized and adapted version of the ECC library available in OpenSSL 0.9.8.

2 Related Work

Since the introduction of elliptic curves to cryptography in 1985 by Koblitz and Miller a vast amount of research has been done. When implementing ECC in software there are several choices to make. We categorize these parameter choices as follows:

- Underlying field type, field representation, and field operation algorithms
- Elliptic curve point representation and point operation algorithms
- Protocol algorithms

The literature contains significant research on each of these issues. As our goal is a generic implementation of ECC, we focus on the optimization of the operations of the underlying field. We think it is easier to optimize a generic ECC implementation over binary fields than over prime fields. As the results in [1] suggest, the efficient use of prime fields requires platform dependent assembly code. Therefore we restrict our attention here to binary fields.

A comprehensive survey of binary field and elliptic curve arithmetic for the NIST recommended elliptic curves was done in [6]. A specialized implementation was done for the field $GF(2^{155})$ [18]. There are also implementations available for constrained devices. PGP was ported to constrained devices in [1] while [23] optimizes an ECC implementation for a Palm PDA. López and Dahab presented a Montgomery field multiplication for binary fields in [14]. They also did research on binary field multiplication [15] and ECC over binary fields [13]. More point multiplication algorithms can be found in [10] and [12]. Further work was done for special field choices like composite fields, e.g., in [2] and [4]. There were, however, recent attacks [3] on these fields. Finally, Solinas developed efficient algorithms for Koblitz curves over binary fields using complex multiplication [22].

3 Arithmetic

3.1 Binary Field $GF(2^m)$

In the following we present new algorithms that are suitable for performing efficient general binary field arithmetic. For the field representation we chose a polynomial basis as it is most widely used and as it yields a simple but fast implementation in software. Let $f(x) = x^m + f'(x)$ be an irreducible binary polynomial of degree m. The elements of $GF(2^m)$ are represented by binary polynomials of degree at most $m-1$. Operations in $GF(2^m)$ are performed modulo $f(x)$. We represent a field element $a(x) = \sum_{i=0}^{m-1} a_i x^i \in GF(2^m)$ as a binary vector $a = (a_{m-1}, \ldots, a_0)$ of length m. As we use word arithmetic, where the register size is noted as W, we also write $A = (A[s-1], \ldots, A[0])$ with $A[i] = (a_{iW+W-1}, \ldots, a_{iW+1}, a_{iW})$ and $s = \lceil m/W \rceil$. Note that $A[s-1]$ is padded on the left with zeros as needed. The expression $A\{i\}$ denotes the sub array $A\{i\} = (A[s-1], \ldots, A[i])$.

There are two main strategies for implementing general binary fields. The first one is to use the Montgomery multiplication and squaring as presented in [9].

The Montgomery multiplication roughly requires two polynomial multiplications but omits the reduction step. The other possibility is to perform field operations in two steps, namely the polynomial operation and reduction. Since we estimate as described later that a general reduction for an arbitrary field polynomial can be done with our new method in time roughly 1/3 of a polynomial multiplication at the cost of memory storage we decided to use this technique.

Addition of field elements is done as word-wise XOR. Since the field size is not fixed we do not know the operand's length at implementation time. We use a jump table to unroll the loop as shown in Algorithm 1 as it gave us a speed-up in our experiments. However, this technique does not necessarily improve running time on all platforms.

Algorithm 1 Addition using jump table

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$ and s words.
OUTPUT: The binary polynomial $c(x) = a(x) \oplus b(x)$.

1: **switch (s)**
2: **case 40:** $C[39] \leftarrow A[39]$ XOR $B[39]$
3: ...
4: **case 1:** $C[0] \leftarrow A[0]$ XOR $B[0]$
5: **break**
6: **default:** do standard addition with loop
7: **end switch**
8: **Return** $c(x)$

We perform division according to [20] and squaring in linear time complexity by a table lookup as presented in [18]. In the following we focus on the multiplication and reduction operation.

Multiplication Modular multiplication is done in two phases. First we compute the polynomial product $c'(x) = a(x) \cdot b(x)$ and then we reduce the result $c(x) \equiv c'(x) \bmod f(x)$. We look at the reduction step in the next section and consider the first step here.

We implemented the Karatsuba Algorithm (KA) and the comb method [15]. The comb method can be implemented as presented in Algorithm 2. Note that W describes the word size, so typical values for W are $8, 16, 32$ and 64. The comb multiplication requires a table of size $2^w s$ where w is the window size. For an efficient implementation one should choose a window size that is a divisor of the word size. We selected $w = 4$ which requires $16s$ table entries for all typical word sizes. Obviously $w = 2$ and $w = 8$ is not optimal for operand sizes as they are used in ECC. We implemented the left shift operation in Steps 1 and 9 as a jump table in a manner similar to the field addition to achieve loop unrolling while still having a general implementation. Note that the left shift of Step 9 is done over the double-sized result operand. Thus, for large word sizes W it might be more efficient to decrease the number of runs of the outer loop in Step 3 by

"emulating" smaller word sizes. For instance, operands of 64-bit words might be considered as operands of twice as many 32-bit words. Then the outer loop is executed only 3 times instead of 7 times whereas the instructions, namely the XOR operation, would still be performed over 64-bit words to utilize the advantage of larger word sizes. However, this might lead to alignment errors at run time on several hardware platforms and thus cannot be seen as a general approach.

Algorithm 2 Comb method with window size $w = 4$

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m - 1$.
OUTPUT: The binary polynomial $c'(x) = a(x) \cdot b(x)$.

1: Compute $B_u(x) = u(x) \cdot b(x)$ for all polynomials $u(x)$ of degree at most 3.
2: $C' \leftarrow 0$
3: **for** $i = W/4 - 1$ down to 0 **do**
4: **for** $j = 0$ to $s - 1$ **do**
5: Let $u = (u_3, u_2, u_1, u_0)$, where u_k is bit $(4i + k)$ of $A[j]$.
6: $C'\{j\} \leftarrow C'\{j\} \oplus B_u$
7: **end for**
8: **if** $i \neq 0$ **then**
9: $C' \leftarrow C'x^4$
10: **end if**
11: **end for**
12: **Return** $c'(x)$

The basic KA works as follows. Consider two operands $A(x)$ and $B(x)$.

$$A(x) = A[1]x^W + A[0],\ B(x) = B[1]x^W + B[0]$$

Let $D_0, D_1, D_{0,1}$ be auxiliary variables with

$$D_0 = A[0]\ B[0], \quad D_1 = B[1]\ B[1]$$
$$D_{0,1} = (A[0] \text{ XOR } A[1])(B[0] \text{ XOR } B[1])$$

Then the polynomial $C(x) = A(x)\ B(x)$ can be calculated in the following way:

$$C(x) = D_1 x^{2W} + (D_{0,1} \oplus D_0 \oplus D_1)x^W + D_0 \tag{1}$$

The KA can easily be expanded in a recursive way, or by splitting the operands into more than two words [24]. For example, to multiply operands of three words using the KA, compute

$$\begin{aligned} C(x) = {} & D_2 x^{4W} + (D_{1,2} \oplus D_1 \oplus D_2)x^{3W} + (D_{0,2} \oplus D_2 \oplus D_0 \oplus D_1)x^{2W} \\ & + (D_{0,1} \oplus D_1 \oplus D_0)x^W + D_0 \end{aligned} \tag{2}$$

where

$$D_0 = A[0]\ B[0], D_1 = A[1]\ B[1], D_2 = A[2]\ B[2]$$

$$D_{0,1} = (A[0] \text{ XOR } A[1])(B[0] \text{ XOR } B[1])$$
$$D_{0,2} = (A[0] \text{ XOR } A[2])(B[0] \text{ XOR } B[2])$$
$$D_{1,2} = (A[1] \text{ XOR } A[2])(B[1] \text{ XOR } B[2])$$

Obviously this can also be done in a recursive manner.

We implemented the KA as follows. First we multiply two words using a slightly modified comb method that does not require a multi-word shift operation. The complexity of the comb method [15] can be generalized to $s\ (\frac{m}{w} + 2^w - w - 1)$ XOR and $w - 1 + 2(\frac{W}{w} - 1)$ SHIFT operations. In our case $s = 1$ and $m = W$. Since we avoid a shift over multiple words this reduces to $\frac{W}{w} + 2^w - w - 1$ XOR and $w + \frac{2W}{w} - 3$ SHIFT operations. Assuming that an XOR and a SHIFT operation needs the same execution time we obtain the following optimum window sizes w. For 8-bit and 16-bit hardware we use a window size of 2, for 32-bit a size of 3,[2] and for 64-bit CPUs a size of 4. We call the corresponding macro `1x1_MUL`. Based on this elementary multiplication we realized hard-coded macros to compute the product of 2, 3, and 4 words[3]. We call these macros `2x2_MUL`, `3x3_MUL` and `4x4_MUL`, which call the `1x1_MUL` macro 3, 6, and 9 times, respectively. The cost of the addition operation is negligible for binary fields and is not considered in the following. Our field multiplication is implemented in a way that different functions can be plugged in for different field sizes. To multiply operands that consist of $s = 1, \ldots, 8$ words, we use specialized methods whereas for operands with $s \geq 9$ words, we use a general approach. The cases $s = 1, \ldots, 4$ are mapped onto the hard coded macros. For $s = 5$ we split the operands into two parts of size 3 and 2 words, respectively. This requires $2 \cdot 6 + 3 = 15$ `1x1_MUL`s. The cases $s = 6, \ldots, 8$ are executed in similar ways. Table 1 presents the cost of the specialized cases in terms of the number of `1x1_MUL`s executed.

Table 1. Cost (in number of `1x1_MUL`s) of KA for n =2-8 words

n	1	2	3	4	5	6	7	8
cost	1	3	6	9	15	18	24	27

For the general case we implemented a recursive method as shown in Algorithm 3. Note that a_i and b_i in Steps 10, 12 and 14 describe the sub-polynomials of a and b, respectively. Clearly, the algorithm could be extended such that the recursion ends in more than the four multiplication macros and the recursive selection handles more cases of s than only those divisible by 2 and 3. We call this the *KA factor basis*. We used a simulation to evaluate the improvements gained by increasing the number of primes in the factor basis: we compared the

[2] Contrary to our earlier statement we use a window size that is not a divisor of the word-size since we are only working on one register.

[3] The method for 4 words has the same complexity as a recursive version computing products of two words but saves some overhead. However, this comes at the cost of a larger code-size.

number of operations for a range of $s = 5, \ldots, 40$ and computed the average cost for the different factor bases. We found that on average a performance gain of 8% is realized when going from the factor basis of 2 to that of 2 and 3, and no performance gain is achieved when adding 5, 7, or 11 to the factor basis. The factors 5, 7, and 11 can optimally be expressed by the factor basis of 2 and 3. For example, the best method to perform KA for 7 words is to split the operand into two portions of 3 and 4 words, respectively. These can optimally be computed by the KA using a prime basis of 2 and 3. Thus these larger factors in the prime basis do not give an additional speed-up. In practice there still might be a performance gain from enlarging the KA factor basis since the recursion overhead is reduced. On the other hand, the code complexity and size increases. We also note that the performance improvement is for the average case. There are some cases, in which a smaller factor basis executes faster. For example, for $s = 15$ words a factor basis of 2 and 3 requires 90 elementary `1x1_MUL` operations while a factor basis of 2 only needs 78 such operations.

Algorithm 3 General Karatsuba Algorithm `GEN_KA(`a`,`b`,`s`)`

INPUT: Binary polynomials $a(x)$ and $b(x)$ of s words.
OUTPUT: The binary polynomial $c'(x) = a(x) \cdot b(x)$.

1: **if** $s = 4$ **then**
2: **Return** `4x4_MUL(a,b)`
3: **else if** $s = 3$ **then**
4: **Return** `3x3_MUL(a,b)`
5: **else if** $s = 2$ **then**
6: **Return** `2x2_MUL(a,b)`
7: **else if** $s = 1$ **then**
8: **Return** `1x1_MUL(a,b)`
9: **else if** $s \bmod 3 = 0$ **then**
10: Split a and b each into three sub polynomials a_i, $i = 1, 2, 3$, of size $s/3$, and b_i, respectively. Perform `GEN_KA(`a_i`,`b_i`,`$s/3$`)` 6 times according to the elementary 3-word KA (2), and put the result together.
11: **else if** $s \bmod 2 = 0$ **then**
12: Split a and b each into two sub polynomials a_i, $i = 1, 2$, of size $s/2$, and b_i, respectively. Perform `GEN_KA(`a_i`,`b_i`,`$s/2$`)` 3 times according to the elementary 2-word KA (1), and put the result together.
13: **else**
14: Split a and b each into two sub polynomials a_i, $i = 1, 2$, of size $\lfloor s/2 \rfloor$ and $\lceil s/2 \rceil$, and b_i, respectively. Perform `GEN_KA(`a_i`,`b_i`,`$\lceil s/2 \rceil$`)` twice and `GEN_KA(`a_i`,`b_i`,`$\lfloor s/2 \rfloor$`)` once according to the elementary 2-word KA (1), and put the result together.
15: **end if**

Reduction Reduction can be performed efficiently for trinomials and pentanomials with middle terms close to each other. All standardized curves use these kinds of field polynomials. It is known that for all binary fields $GF(2^m)$ up

to at least $m = 1000$ there exists such an irreducible polynomial [11]. While reduction can be hard-coded for a fixed field polynomial and executed at negligible cost, this is not the case for a generic implementation.

As before we decided to offer different reduction methods which can be plugged in for different fields. We offer a reduction method for trinomials as well as for pentanomials which performs a word-wise reduction for general trinomials and pentanomials, respectively [6]. Furthermore we implemented a general version that can handle arbitrary field polynomials. However, it becomes very expensive for dense irreducible polynomials. It is clear that even the specialized methods for trinomials and pentanomials cannot be nearly as efficient as hard coded methods for a given irreducible polynomial. Another universal method to perform reduction is division with remainder which usually is expensive.

We propose a third reduction method using a window technique as described in Algorithm 4. This algorithm contains a precomputation step in which all possible window values are computed and stored in a table, and a computation step in which the polynomial is reduced by one window segment at a time. The algorithm can be best explained with the following example. Let $f(x) = x^m + f'(x) = x^{163} + x^7 + x^6 + x + 1$, $W = 32$, $s = 6$, and let $\ll t$ and $\gg t$ be a shift to the left or right by t bit positions, respectively. Furthermore, let $PreF_i$ be the table entry at index i. First, we insert the polynomial $f'(x)$ into the table at index $2^{163 \bmod 32}$. Since we will perform shifts of this table entry to the right in order to obtain further values we attach a zero-word below the least significant bit, i.e., we store $PreF_8 = f'(x) \cdot x^W$. The remaining table entries can now be computed as $PreF_1 = PreF_8 \gg 3$, $PreF_2 = PreF_1 \ll 1$, $PreF_3 = PreF_2 \oplus PreF_1$, and so on. Further speed-up can be achieved by computing table entries $PreF_{j \cdot t}$ with $j \in \{1, \ldots, W/w\}$ and $t \in \{1, \ldots, 2^w - 1\}$ where w is the window size. This avoids shifting operations in the computation step. In the computational phase the polynomials are reduced by a window segment in each step. Let $c(x)$ be a polynomial of order larger than 162, and let $(c_{iW+jw+w-1}, \ldots, c_{iW+jw})$ be a window of $C[i]$ with $iW + jw \geq 163$. Then this window segment can be reduced by XORing $C\{i - s\}$ with $PreF_{(j+1)(iW+jw+w-1,\ldots,iW+jw)_2}$, where $C\{i - s\}$ denotes the sub array $C\{i-s\} = (C[i], \ldots, C[i-s])$. The addition in Step 6 can be done efficiently by only performing an XOR operation for values not equal to 0. The last word $C[s-1]$ is reduced separately to take into account that only a part of it is reduced, namely $(c_{(s-1)W+W-1}, ..., c_{(s-1)W+(m \bmod W)}, 0, ..., 0)$. Note that in the last step we XOR $(PreF_{j \cdot t} \gg W)$ to C.

The technique is very similar to the comb method for polynomial multiplication [15]. Each table entry $PreF_u(x)$ requires $s + 1$ words. Thus the memory requirement is $(W/w)(2^w - 1)(s + 1)$ words. Algorithm 4 uses window size $w = 8$ as it is most suitable for a server application and needs a memory size of $255(W/8)(s + 1)$ words[4]. The precomputation step is done only once per field and can be done off-line. Our reduction method does not require any shift operations. Assuming that the precomputation is done off-line the method requires

[4] A window size of $w = 8$ requires for a 32-bit CPU ($W = 32$) and a 163-bit curve ($s = 6$) 28 KB while a window size of $w = 16$ needs 3.5 MB.

$(m/w)(s+1)$ XORs. Thus a reduction step costs $(m/8)(s+1)$ XOR operations in our case. The comb method requires $s(m/4+11)$ XORs and $3+2(W/4-1)$ SHIFTs over multiple words [15]. Hence the cost of a reduction step for arbitrary field polynomials is less than 1/2 of a comb polynomial multiplication. For typical field sizes of $m = 163$ our reduction method has a running time of approximately 1/3 of a comb multiplication. We expect this to be the fastest general reduction algorithm at the cost of memory storage.

Further improvements could be made by doing a partial reduction [5] as follows. Usually we reduce an operand $c'(x)$ of size $2s$ to $c(x)$ such that $degree(c) < m$. However, we can save some calculations by reducing $c'(x)$ to $c''(x)$ such that $degree(c''(x)) < sW$. The binary field operations are able to handle this operand size without any further cost. Algorithm 4 can easily be adjusted to this fact by omitting Steps $9-12$.

Algorithm 4 Window reduction method with window size $w = 8$

INPUT: A binary polynomial $c'(x)$ of $2s$ words and an irreducible binary polynomial $f(x) = x^m + f'(x)$ of s words.
OUTPUT: The binary polynomial $c(x) \equiv c'(x) \bmod f(x)$.

1: Precompute $PreF_u$ for all $u = jt$ with $j \in \{1, \ldots, W/8\}$ and $t \in \{1, \ldots, 2^8 - 1\}$:
   $PreF_1 = (f'(x) \cdot x^W) \gg (m \bmod W)$
   $PreF_{2^v} = PreF_{2^{v-1}} \ll 1$ for $0 < v < 8$
   $PreF_t = PreF_{2^v} \oplus PreF_{t-2^v}$ for $2^{v+1} < t < 2^v, 0 < t < 2^8$
   $PreF_{jt} = PreF_t << 8(j-1)$ for $j \in \{2, ..., W/8\}$ and $t \in \{1, \ldots, 2^8 - 1\}$
2: $C \leftarrow C'$
3: **for** $i = 2s - 1$ down to s **do**
4: **for** $j = W/8 - 1$ down to 0 **do**
5: Let $t = (t_7, t_6, t_5, t_4, t_3, t_2, t_1, t_0)$, where t_k is bit $(8j + k)$ of $C[i]$.
6: $C\{i - s\} = C\{i - s\} \oplus PreF_{j \cdot t}$
7: **end for**
8: **end for**
9: **for** $j = W/8 - 1$ down to 0 **do**
10: Let $t = (t_7, t_6, t_5, t_4, t_3, t_2, t_1, t_0)$, where t_k is bit $(8j + k)$ of $C[s - 1]$ if $(8j + k) \geq (m \bmod W)$ and $t_k = 0$ otherwise.
11: $C\{0\} = C\{0\} \oplus (PreF_{j \cdot t} \gg W)$
12: **end for**
13: **Return** $c(x)$

3.2 Elliptic Curve $E(GF(2^m))$

We chose standard algorithms to implement the general elliptic curves. We use the projective Montgomery point multiplication as introduced in [14]. We also implemented standard point addition and point doubling [7].

Koblitz Curves We implemented Koblitz Curves as described in [22]. We chose the windowed NAF technique for point multiplication using projective coordinates as introduced in [13]. Most of the values needed for the computation can be computed on the fly at initialization time. However, obtaining the window coefficients α_i [22] requires operations such as complex number arithmetic that are not supported by our software library. There are two kinds of Koblitz curves, namely for curve parameter $a = 0$ and $a = 1$. We decided to support window sizes of $w = 5$ and $w = 6$. The windowed NAF technique performs a point multiplication at the cost of $2^{w-2} - 1 + \frac{m}{w+1}$ point additions for fields $GF(2^m)$. Thus for $m < 336$ we use a window size of $w = 5$ whereas for $m \geq 336$ we choose $w = 6$. A window size of $w = 4$ is efficient for $m < 120$ and a window size of $w = 7$ for $m > 896$. Since these field sizes are not relevant for ECC we do not support these window sizes. We precomputed all possible window coefficient values α_i for the four combinations off-line, namely $a = 0$ and $a = 1$ combined with $w = 5$ and $w = 6$. The memory storage for the precomputed values is negligible and would easily fit into the storage of a smart card.

4 Implementation

In this section we give timings for our implementation and analyze the results. We did our timings for the NIST recommended binary fields and curves over $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$ and $GF(2^{571})$ [16]. We considered general curves and Koblitz curves which are denoted as B-163, B-233, B-283, B-409, B-571 and K-163, K-233, K-283, K-409, and K-571, respectively. The fields of bit-size 163, 283, and 571 use pentanomials while fields of bit-size 233 and 409 use trinomials as field polynomials. We compiled our code on a SPARC 32-bit and 64-bit workstation and on an Intel workstation. The SPARC timings were done on a 900 MHz UltraSPARC III CPU running Solaris 9, and gcc version 3.1. The UltraSPARC III is a 64-bit CPU which can also emulate a 32-bit mode. The Intel timings were performed on a 1 GHz Pentium III and gcc version 2.95.3 under Linux. We implemented all algorithms in C and did not use any assembly code.

Table 2 presents our timings of the field operations. The field operations always include the reduction step which is performed by the word-wise method for general trinomials as well as general pentanomials. It is worth noticing that a squaring, which is usually considered to have little cost, is no longer negligible for unknown field polynomials. The cost is especially considerable for small field sizes.

We experienced in our tests that the running time of the comb method is very dependent on the hardware platform and even the compiler. While KA performs similarly on different platforms we were unable to predict the running time of the comb method. It also seemed difficult to implement the comb method in a general fashion. We did some testings by implementing a special comb method for 6 words as needed for B-163. We unrolled all the loops, and used special addition and shifting macros for the given operand's length. The obtained timings are

Table 2. Timings in μs for one field operation (for trinomials and pentanomials as field polynomial)

Field Size	Operation		SPARC 32-bit 900 MHz	SPARC 64-bit 900 MHz	Intel 1 GHz
163	Multiplication	Comb Method	3.9	3.4	2.8
		Karatsuba	2.3	1.3	1.9
	Reduction		0.7	0.4	0.4
	Squaring		0.8	0.5	0.5
233	Multiplication	Comb Method	5.8	4.8	3.8
		Karatsuba	2.8	1.5	2.9
	Reduction		0.3	0.2	0.3
	Squaring		0.5	0.4	0.4
283	Multiplication	Comb Method	6.6	5.8	4.5
		Karatsuba	4.5	3.3	4.4
	Reduction		0.7	0.5	0.5
	Squaring		1.0	0.8	0.6
409	Multiplication	Comb Method	10.7	8.9	6.9
		Karatsuba	8.5	4.7	8.1
	Reduction		0.5	0.3	0.4
	Squaring		0.8	0.6	0.6
571	Multiplication	Comb Method	17.7	12.8	10.6
		Karatsuba	16.4	7.0	13.4
	Reduction		1.3	0.9	0.9
	Squaring		1.8	1.2	1.2

comparable to the KA times. However, we did not find a way to implement the comb method faster in a general way. Having specialized methods for each size results in large and complex code, while plugging in different methods for different cases requires expensive function calls and cannot be done as compile-time macros. However, it seems that the comb method outperforms the KA for large operand sizes. For a server implementation one could implement both the comb method and KA, do some test runs when a new field comes into use and plug in the faster method.

Now we want to point the reader's attention to the 64-bit multiplication. While the KA performs nearly twice as fast as the 32-bit version, the comb method gets only slightly faster. Our `1x1_MUL` for 64-bit requires about 65% more operations than the 32-bit version. However, the number of multiplications required for the KA applied to operands of half the size decreases by a factor of up to 3. Thus the KA becomes almost twice as fast, i.e., by a factor of around $3/1.65 = 1.82$. However, the comb method requires $W/4 - 1$ left shifts of the double-sized result operand. Since the number of shifts doubles and the number of words is halved the overall computational cost remains unimproved. For instance, if the operand's length is s words on a 32-bit machine and $s' = s/2$ words on a 64-bit computer, the number of shifts evaluates to $3s$ and $7s' = 3.5s$, respectively. Since shifting has linear time complexity, i.e. a shift over $2t$ words

takes twice as long as a shift over t words, the number of executable CPU instructions roughly remains the same. Thus there is only little speed gain due to the accelerated XOR operations for the comb method on a 64-bit CPU.

We compared our numbers to NTL 5.3 [21] which is considered to be one of the fastest number arithmetic libraries publicly available. The results for the Intel platform are presented in Table 3. Both multiplication and squaring include the modulo reduction. Note that we compare the timings to our KA implementation although on the Intel platform it is slower than the comb method for large field sizes. Our results are considerably faster for the essential field multiplication. One can guess that this is due to our polynomial multiplication method.

Table 3. Timings in μs for one field operation on 1 GHz Intel

Field Size	Operation	NTL	GEN_KA
163	Multiplication	4.7	1.9
	Reduction	0.8	0.4
	Squaring	1.1	0.5
233	Multiplication	5.5	2.9
	Reduction	1.0	0.3
	Squaring	1.1	0.4
283	Multiplication	9.7	4.4
	Reduction	1.1	0.5
	Squaring	1.8	0.6
409	Multiplication	14.6	8.1
	Reduction	1.1	0.4
	Squaring	1.3	0.6
571	Multiplication	27.4	13.4
	Reduction	1.2	0.9
	Squaring	3.1	1.2

Table 4 presents the timings for the elliptic curve point multiplication. For each platform, the first column is based on the KA while the second one uses the comb method. When comparing our timings to the numbers for fixed fields presented in [6] one can see that our flexible implementation still is slower than such a specialized implementation.

5 Conclusions

In this paper we presented an industry implementation of ECC as part of OpenSSL. Our implementation takes a general approach to implement binary field arithmetic and does not depend on any choice of field size or field polynomial. We showed that we achieve running times that are in the range of previously reported implementation results for fixed field sizes. Furthermore we showed that our implementation is faster than other implementations for arbitrary field sizes.

Table 4. Timings in *ms* for one point multiplication

	SPARC 32-bit 900 MHz		SPARC 64-bit 900 MHz		Intel 1 GHz	
Curve	KA	Comb	KA	Comb	KA	Comb
B-163	3.0	4.9	1.8	4.1	2.3	3.6
B-233	4.8	8.7	2.8	7.3	4.7	6.4
B-283	10.0	13.9	6.8	12.0	9.5	9.7
B-409	24.9	30.0	14.1	24.3	23.6	19.8
B-571	66.6	71.5	30.3	51.3	54.3	44.9
K-163	1.6	2.0	1.0	1.7	1.2	1.4
K-233	2.2	3.3	1.5	2.6	2.0	2.5
K-283	4.6	5.4	3.1	4.3	3.7	3.8
K-409	9.6	10.9	5.4	7.7	8.0	7.2
K-571	22.5	23.8	11.4	15.9	17.2	15.3

Acknowledgments

We thank the anonymous referees for their helpful and detailed remarks. We are also grateful to the people of Sun Microsystems Laboratories at Mountain View for their support.

References

1. M. Brown, D. Cheung, D. Hankerson, J. L. Hernandez, M. Kirkup, and A. Menezes. PGP in Constrained Wireless Devices. *Proceedings of the 9th USENIX Security Symposium*, 2000.
2. E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem and J. Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. *Advances in Cryptology – ASIACRYPT '96*, LNCS 1163, Springer-Verlag, 65-76, 1996.
3. P. Gaudry, F. Hess, and N. Smart. Constructive and Destructive Facets of Weil Descent on Elliptic Curves. *Journal of Cryptology*, 15, 19-46, 2002.
4. J. Guajardo and C. Paar. Efficient algorithms for elliptic curve cryptosystems. *Advances in Cryptology – CRYPTO '97*, LNCS 1294, Springer-Verlag, 342-356, 1997.
5. N. Gura, H. Eberle, and S. Chang Shantz. Generic Implementations of Elliptic Curve Cryptography using Partial Reduction. *9th ACM Conference on Computer and Communications Security*, 2002.
6. D. Hankerson, J. L. Hernandez and A. Menezes. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. *Cryptographic Hardware and Embedded Systems, CHES 2000*, LNCS 1965, Springer-Verlag, 1-24, 2000.
7. IEEE P1363. *Standard Specifications for Public-Key Cryptography*, 2000.
8. ISO/IEC 15946. *Information Technology – Security Techniques – Cryptographic Techniques Based on Elliptic Curves*, 1999.
9. C. Koç and T. Acar. Montgomery multiplication in $GF(2^k)$. *Designs, Codes and Cryptography*, 14, 57-69, 1998.

10. K. Koyama and Y. Tsuruoka. Speeding up elliptic curve cryptosystems by using a signed binary window method. *Advances in Cryptology – Crypto '92*, LNCS 740, Springer-Verlag, 345-357, 1993.
11. R. Lidl and H. Niederreiter. *Introduction to Finite Fields and their Applications, Revised Edition*. Cambridge Unversity Press, Cambridge, United Kingdom, 1994.
12. C. Lim and P. Lee. More flexible exponentiation with precomputation. *Advances in Cryptology - Crypto '94*, LNCS 839, Springer-Verlag, 95-107, 1994.
13. J. López and R. Dahab. Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$. *Selected Areas in Crytography - SAC '98*, LNCS 1556, Springer-Verlag, 201-212, 1999.
14. J. López and R. Dahab. Fast multiplication on Elliptic Curves over $GF(2^n)$ without Precomputation, *Cryptographic Hardware and Embedded Systems-CHES '99*, LNCS 1717, Springer-Verlag, 316-327, 1999.
15. J. López and R. Dahab. High-speed Software Multiplication in $\mathbb{F}_{2^m}$. *Indocrypt 2000*, LNCS 1977, Springer-Verlag, 203-212, 2000.
16. National Institute of Standards and Technolgy. *Recommended Elliptic Curves for Federal Government Use*, May 1999, available from http://csrc.nist.gov/encryption.
17. OpenSSL, http://www.openssl.org.
18. R. Schroeppel, H. Orman, S. O'Malley and O. Spatscheck. Fast Key Exchange with Elliptic Curve Systems. *Advances in Cryptology - Crypto '95*, LNCS 963, Springer-Verlag, 43-56, 1995.
19. R. Schroeppel, C. Beaver, R. Gonzales, R. Miller, and T. Draelos. A Low-Power Design for an Elliptic Curve Digital Signature Chip. Presented at *Cryptographic Hardware and Embedded Systems (CHES) 2002*.
20. S. Shantz. From Euclid's GCD to Montgomery Multiplication to the Great Divide, preprint, 2000.
21. V. Shoup. NTL: A Library for doing Number Theory, available from http://www.shoup.net/ntl/.
22. J. A. Solinas. Efficient Arithmetic on Koblitz Curves. *Designs, Codes and Cryptography*, 19(2/3), 195-249, 2000.
23. A. Weimerskirch, C. Paar, and S. Chang Shantz. Elliptic Curve Cryptography on a Palm OS Device. *The 6th Australasian Conference on Information Security and Privacy (ACISP 2001)*, LNCS 2119, Springer-Verlag, 502-513, 2001.
24. A. Weimerskirch and C. Paar. Generalizations of the Karatsuba Algorithm for Polynomial Multiplication. *Technical Report*, Ruhr-University Bochum, 2002, Available from http://www.crypto.rub.de.