Impact of Optimized Operations AB,AC and AB+CD in Scalar Multiplication over Binary Elliptic Curve

Christophe Negre, Jean-Marc Robert

Impact of Optimized Operations AB, AC and $AB + CD$ in Scalar Multiplication over Binary Elliptic Curve

C. Negre[1,2,3] and J.-M. Robert[1,2,3]

[1] Team DALI, Université de Perpignan, France
[2] LIRMM, UMR 5506, Université Montpellier 2, France
[3] LIRMM, UMR 5506, CNRS, France

Abstract. A scalar multiplication over a binary elliptic curve consists of a sequence of hundreds of multiplications, squarings and additions. This sequence of field operations often involves a large amount of operations of type AB, AC and $AB + CD$. In this paper, we modify classical polynomial multiplication algorithms to obtain optimized algorithms which perform these particular operations AB, AC and $AB + CD$. We then present software implementation results of scalar multiplication over binary elliptic curve over two platforms: Intel core 2 duo and Intel core i5. These experimental results show some significant improvements due to the proposed optimizations and our results are better than the recently published implementation results [12] of double-and-add scalar multiplication.

Keywords. Optimized operations (AB, AC) and $AB + CD$, double-and-add, scalar multiplication, software implementation, carry less multiplication.

1 Introduction

Finite field arithmetic is widely used in elliptic curve cryptography (ECC) [10,7] and coding theory [2]. The main operation in ECC is the scalar multiplication which is computed as a sequence of multiplications and additions in the underlying field [3,4]. Efficient implementations of these sequences of finite field operations are thus crucial to get efficient cryptographic protocol.

We will focus here on the special case of software implementation of scalar multiplication on elliptic curve defined over an extended binary field $\mathbb{F}_{2^m}$. An element in $\mathbb{F}_{2^m}$ is a binary polynomial of degree at most $m-1$. In practice m is a prime integer in the interval $[160, 600]$. An addition and a multiplication of field elements consist of a regular binary polynomial addition and multiplication performed modulo the irreducible polynomial defining $\mathbb{F}_{2^m}$. An addition and a reduction are in practice faster than a multiplication of degree m polynomial. Specifically, an addition is a simple bitwise XOR of the coefficients: this consists, in software, of computing several independent word bitwise XORs (WXOR). Concerning the reduction, when the irreducible polynomial which defines the field $\mathbb{F}_{2^m}$ is sparse, reducing a polynomial can be expressed as a number of word shifts and word XORs.

Until the end of 2009 the fastest algorithm for software implementation of polynomial multiplication was the Comb method of Lopez and Dahad [9]. This method essentially uses look-up tables, word shifts (Wshift), ANDs and XORs. One of the most recent implementation based on this method was done by Avanzi and Thériault in [1] on an Intel core 2 duo. But, since the introduction by Intel of a new carry less multiplication instruction on the new processors i3, i5 and i7, the authors in [12] has shown that the polynomial multiplication based on Karatsuba method [11] outperforms the former approaches based on Lopez and Dahab multiplication. In the sequel we will consider implementations on two platforms: processor without carry less multiplication (Intel core 2 duo) and processor i5 which has such instruction.

Our contributions. We investigate in this paper some optimizations of the operations AB, AC and $AB + CD$. The fact that we can optimize two multiplications AB, AC which have a common input A, is well known, it was for example noticed in [1]. Indeed, since there is a common input A, the computations depending only on A in AB and AC can be shared.

We also investigate a new optimization based on $AB + CD$. In this situation, we will show that we can save in Lopez-Dahab polynomial multiplication algorithm $60N$ WShifts and $30N$ WXORs if the inputs are stored on N computer words. We will also show that this approach can be adapted to the case of Karatsuba multiplication and evaluate the resulting complexity.

We present implementation results of scalar multiplication which involve the previously mentioned optimizations. The reported results on the Intel core 2 duo were obtained using Lopez-Dahab polynomial multiplication for field multiplication, and the reported results on the Intel core i5 were obtained with Karatsuba multiplication.

Organization of the paper. We review in Section 2 the best known algorithms for software implementation of polynomial multiplication of size $m \in [160, 600]$. We then present in Section 3 optimized versions of these algorithms for the operations AB, AC and $AB + CD$. In Section 4, we describe how to use the proposed optimizations in a scalar multiplication and give implementation results obtained on an Intel core 2 duo and on an Intel core i5. Finally, in Section 5 we give some concluding remarks.

2 Review of multiplication algorithm

The problem considered in this section is to compute efficiently a multiplication in a binary field $\mathbb{F}_{2^m}$. A field $\mathbb{F}_{2^m}$ is defined as the set of binary polynomials modulo an irreducible polynomial $f(x) \in \mathbb{F}_2[x]$ of degree m. Consequently, a multiplication in $\mathbb{F}_{2^m}$ consists to multiply two polynomials of degree at most $m-1$ and reduce the product modulo $f(x)$. The field considered here are described in Table 1 and are suitable for elliptic curve cryptography. The irreducible polynomials in Table 1 have a sparse form. This implies that the reduction can be expressed as a number of shifts and additions (the reader may refer for example to [4] for further details).

We the focus on efficient software implementation of binary polynomial multiplication: we review the best known algorithms for polynomial of cryptographic size. An element $A = \sum_{i=0}^{m-1} a_i x^i \in \mathbb{F}_2[x]$ is coded over $N = \lceil m/64 \rceil$ computer words of size 64 bits $A[0], \ldots, A[N-1]$. In the sequel we will often use a nibble decomposition A:

$$A = \sum_{i=0}^{n-1} A_i x^{4i}$$

where $\deg A_i < 4$ and $n = \lceil m/4 \rceil$ is the nibble size of A. In Table 1 we give the value of N and n for the field sizes $m = 233$, 409 and 503 considered in this paper.

Table 1. Irreducible polynomials and word/nibble sizes of field elements

m the field degree	Irreducible polynomial	N (64bit word size)	n (nibble size)
233	$x^{233} + x^{74} + 1$	4	59
409	$x^{409} + x^{87} + 1$	7	103
503	$x^{503} + x^3 + 1$	8	125

2.1 Comb Multiplication

One of the best known methods for software multiplication of two polynomials A and B was proposed by Lopez and Dahab in [9]. This algorithm is generally referred as the left-to-right comb method with window size w. We present this method for the window size $w = 4$ since this is the best approach in practice. This method first computes a table T containing all products $u \cdot A$ for $u(x)$ of degree < 4. The second input B is decomposed into 64-bit words and nibbles as follows

$$B = \sum_{j=0}^{N-2} \sum_{k=0}^{15} B_{16j+k} x^{64j+4k} + \sum_{k=0}^{n-16(N-1)-1} B_{16(N-1)+k} x^{64(N-1)+4k} \text{ where } \deg B_{16j+k} < 4$$

Then the product $A \times B$ is expressed as follows by expanding the above expression of B

$$\begin{aligned} A \cdot B &= A \cdot (\textstyle\sum_{j=0}^{N-2} \sum_{k=0}^{15} B_{16j+k} x^{64j+4k} + \sum_{k=0}^{n-16(N-1)-1} B_{16(N-1)+4k} x^{64(N-1)+4k}) \\ &= \textstyle\sum_{j=0}^{N-2} \sum_{k=0}^{15} (A \cdot B_{16j+k} x^{64j+4k}) + \sum_{k=0}^{n-16(N-1)-1} (A \cdot B_{16(N-1)+k}) x^{64(N-1)+4k} \\ &= \textstyle\sum_{k=0}^{n-16(N-1)-1} x^{4k} \left(\sum_{j=0}^{N-1} A \cdot B_{16j+k} x^{64j} \right) \\ &\qquad + \textstyle\sum_{k=n-16(N-1)}^{15} x^{4k} \left(\sum_{j=0}^{N-2} (A \cdot B_{16j+k} x^{64j}) \right). \end{aligned}$$

The above expression can be computed through a sequence of accumulations $R \leftarrow R + T[B_{16j+k}]x^{64j}$, corresponding to the terms $A \cdot B_{16j+k} x^{64i}$, followed by multiplications by x^4. This leads to Algorithm 1 for a pseudo-code formulations and Algorithm 6 in the appendix for a C-like code formulation.

Algorithm 1 CombMul(A,B)

Require: Two binary polynomials $A(x)$ and $B(x)$ of degree $< 64N - 4$, and $B(x) = \sum_{i=0}^{N-1} \sum_{k=0}^{15} B_{16j+k} x^{4k+64j}$ is decomposed in words and nibbles.
Ensure: $R(x) = A(x) \cdot B(x)$
  // Computation of the table T containing $T[u] = u(x) \cdot A(x)$ for all u such that $\deg u(x) < 4$
  $T[0] \leftarrow 0$;
  $T[1] \leftarrow 0$;
  for k **from** 1 **to** 7 **do**
    $T[2k] \leftarrow T[k] \cdot x$;
    $T[2k+1] \leftarrow T[2k] + A$;
  end for
  // right-to-left shifts and accumulations
  $R \leftarrow 0$
  for k **from** 15 **downto** 0 **do**
    for j **from** $N-1$ **downto** 0 **do**
      $R \leftarrow R + T[B_{16j+k}]x^{64j}$
    end for
    $R \leftarrow R \cdot x^4$
  end for

Complexity. We evaluate the complexity of the corresponding C-like code (Algorithm 6) of the `CombMul` algorithm in terms of the number of 64-bit word operations (WXOR, WAND and WShift). We do not count the operations performed for the loop variables $i, j, \ldots$. Indeed when all the loops an unrolled, these operations can be precomputed. We have separated the complexity evaluation of the `CombMul` algorithm into three parts: the computation of the table T, the accumulations $R \leftarrow R + T[B_{16j+k}]x^{64j}$ and the shifts $R \leftarrow R \cdot x^4$ of R.

- *Table computation.* The loop on k is of length 7, and performs one WXOR and one WShift plus $2(N-1)$ WXORs and $2(N-1)$ WShifts in the inner loop on i.
- *Shifts by* 4. There are two overlapped loops: the one on k is of length 15 and the loop on i is of length $2N$. The loops operations consist of two WShifts and one WXOR.
- *Accumulations.* The number of accumulations $R \leftarrow R + T[B_{16j+k}]x^{64j}$ is equal to n, the nibble length of B. This results in nN WXOR, n WAND and $n - N$ WShift operations, since a single accumulation $R \leftarrow R + T[B_{16j+k}]x^{64j}$ requires N WXOR, one WAND and one WShift (unless for $k = 0$).

As stated in Table 2, the total number of operations is equal to $nN + 44N - 13$ WXORs, n WANDs and $n + 73N - 13$ WShifts.

2.2 Karatsuba multiplication

We review in this section the Karatsuba approach for binary polynomial multiplication. For the sake of simplicity we assume that m is even. This approach consists to first split A and B in two halves

Table 2. Complexity of the C code of the Comb multiplication

Operation	#WXOR	#WShift	#WAND
Table T	$14N-7$	$14N-7$	0
$R \leftarrow R + T[B_{16j+k}]x^{64j}$	nN	$n-N$	n
Shift $R \leftarrow R << 4$	$30N$	$60N$	0
Total	$nN+44N-7$	$n+73N-7$	n

$A = A_0 + x^{m/2}A_1$ and $B = B_0 + x^{m/2}B_1$ and then re-express the product $A \times B$ in terms of three polynomial multiplications of half size:

$$\begin{array}{l} R_0 = A_0B_0,\ R_1 = A_1B_1,\ R_2 \quad = \quad (A_0+A_1)(B_0+B_1), \\ C = R_0 + x^{m/2}(R_0+R_1+R_2) + x^m R_1\,. \end{array} \tag{1}$$

We consider here two variants of the Karatsuba approach.

`KaratRec` **approach (Algorithm 2).** In this case the inputs A and B are supposed to be of size $64N$ bits where $N = 2^s$ and packed in an array of N computer words. The three products R_0, R_1 and R_2 are computed recursively until we reach inputs of size one computer word. Then the word products are computed with the a `Mult64`. We further assume that this `Mult64` operation is performed using a single processor instruction: this is the case of the Intel cores i3, i5 and i7.

Algorithm 2 `KaratRec`(A,B)

Require: A and B on $n = 2^s$ computer words.
Ensure: $R = A \times B$
  if $n = 1$ **then**
    return ($Mult64(A, B)$)
  else
    // Split in two parts of word size $N/2$.
    $A = A_0 + x^{64N/2}A_1$
    $B = B_0 + x^{64N/2}B_1$
    // Recursive multiplication
    $R_0 \leftarrow$ `KaratRec`$(A_0, B_0, N/2)$
    $R_1 \leftarrow$ `KaratRec`$(A_1, B_1, N/2)$
    $R_2 \leftarrow$ `KaratRec`$(A_0 + A_1, B_0 + B_1, N/2)$
    // Reconstruction
    $R \leftarrow R_0 + (R_0 + R_1 + R_2)X^{64N/2} + R_1X^{64N}$
    return (R)
  end if

Complexity of `KaratRec` approach. We briefly compute the complexity of the `KaratRec` algorithm in terms of the number of WXOR and `Mult64` operations. One single recursion of the Karatsuba formula with inputs of word size N requires N WXORs for the additions A_0+A_1 and B_0+B_1, and $5N/2$ WXORs for the reconstruction of R. We obtain the recursive complexity given in the left side of (2). We rewrite the complexity in the non-recursive form given in the right side of (2).

$$\begin{array}{l} \left\{ \begin{array}{l} \#WXOR(N){=}4N + 3\#WXOR(N/2), \\ \#WXOR(1){=}0. \end{array} \right. \implies \#WXOR(N) = 8N^{\log_2(3)} - 8N \\ \left\{ \begin{array}{l} \#Mult64(N){=}3\#Mult64(N/2), \\ \#WXOR(1){=}1. \end{array} \right. \implies \#Mult64(N) = N^{\log_2(3)}. \end{array} \tag{2}$$

`KaratComb` **approach (Algorithm 7).** We apply the formula of Karatsuba only once: the two inputs A and B are split into halves of nibble size $n/2$, i.e., $B = A_0 + x^{4n/2}A_2$ and $B = B_0 + x^{4n/2}B_2$. The

remaining three products of (1) are performed with the `CombMul` algorithm

$$R_0{=}\texttt{CombMul}(A_0, B_0),\ R_1{=}\texttt{CombMul}(A_0 + A_2, B_0 + B_2),\ R_2{=}\texttt{CombMul}(A_2, B_2),$$
$$R = R_0 + x^{4n/2}(R_0 + R_1 + R_2) + x^{4n}R_2 \quad .$$

The technical part here is in the splitting phase and the reconstruction phase. In the appendix we provide a C-like version of this algorithm which includes the hidden operations involved in the splitting and the reconstruction.

Complexity of the `KaratComb` approach. The complexity is deduced from the complexity of each step established in Algorithm 7 and from Table 2 for the three multiplications `CombMul` and the word and nibble sizes $N/2$ and $n/2$. We obtain the following complexities

$$\begin{cases} \#WXOR(n) = nN/4 + 9N + 8, \\ \#WShift(n) = n/2 + 85N/2 + 5, \\ \#WAND(n) = n/2 + 2. \end{cases}$$

3 Optimization of the operations $AB + CD$ and AB, AC

We present in this section our main building blocks for the optimization of software implementation of elliptic curve scalar multiplication. The main idea is that the scalar multiplication involves operations of type $AB + CD$ or AB, AC. In such operations $AB + CD$ and AB, AC some computations can be saved resulting in a more efficient software implementation. This idea was previously noticed for example in [1] for AB, AC for the `CombMul` algorithm. We extend this idea to the variants based on Karatsuba multiplication. We also study the optimization based on the operation $AB + CD$ in the case of `CombMul` algorithm and in the case of the variants of Karatsuba multiplication.

3.1 Optimizations of $AB + CD$ and AB, AC in the CombMul approach

Optimization AB, AC in the `CombMul` algorithm. The fact that we have to compute two multiplications with the same operand A, implies that the table T in the `CombMul` algorithm, which contains the products $T[u] = u \cdot A$, can be computed only once for the two multiplications AB and AC. This saves $14N - 7$ WXORS and $14N - 7$ Shifts operations in the computation of AC. The resulting complexity of the `CombMul_ABAC` algorithm is shown in Table 3. The `CombMul_ABAC` algorithm is described in the appendix.

Optimization $AB + CD$ in the `CombMul` algorithm. We optimize the operation $AB + CD$ by performing the final addition $(AB) + (CD)$ during the accumulation step of the `CombMul` algorithm. Specifically, we keep the table computation stage $T[u] = u \cdot A$ and $S[u] = u \cdot C$ for u of degree < 4 unchanged. But we accumulate $T[B_{16j+k}]$ and $S[D_{16j+k}]$ in the same variable $R \leftarrow R + (T[B_{16j+k}] + S[B_{16j+k}])x^{64j}$. The shifts by 4 are then performed only on R.

The complexity of Algorithm 3 can be easily deduced from the complexity of the `CombMul` algorithm (Table 2):

- We have in the `CombMul_ABplusCD` algorithm two table computations which contribute to twice the complexity of the table computation in Table 2.
- The accumulations $R \leftarrow R + T[A_i]x^{64j} + S[C_i]x^{64j}$ also contribute to twice the complexity of the accumulation step in Table 2.
- We have the same amount of shifts $R \leftarrow R \cdot x^4$ as in the `CombMul` algorithm.

The resulting complexity is given in Table 3.

3.2 Optimizations $AB + CD$ and AB, AC in the KaratRec approach

The optimization based on AB, AC can be extended to the KaratRec algorithm. Indeed the recursive splitting and the addition of the two halves $A_0 + A_1$ can be performed only once for the polynomial A. This approach is described in Algorithm 5.

Algorithm 3 `CombMul_ABplusCD`(A,B)

Require: Four binary polynomials A, B, C and D of degree $< 64N-4$, and $B(x) = \sum_{j=0}^{N-1}\sum_{k=0}^{15} B_{16j+k}x^{4k+64j}$ with $\deg B_{16j+k} < 4$ and $D(x) = \sum_{j=0}^{N-1}\sum_{k=0}^{15} D_{16j+k}x^{4k+64j}$ with $\deg D_{16j+k} < 4$
Ensure: $R(x) = A(x)\cdot B(x) + C(x)\cdot D(x)$
// Computation of the table T and S such that $T[u] = u(x)\cdot A(x)$ and $S[u] = u(x)\cdot B(x)$ for all $\deg u(x) < 4$
$T[0] \leftarrow 0$; $S[0] \leftarrow 0$;
$T[1] \leftarrow A$; $S[1] \leftarrow C$;
for k **from** 1 **to** 7 **do**
  $T[2k] \leftarrow T[k]\cdot x$; $S[2k] \leftarrow S[k]\cdot x$;
  $T[2k+1] \leftarrow T[2k] + A$; $S[2k+1] \leftarrow S[2k] + C$;
end for
// right-to-left shift Comb multiplication
$C \leftarrow 0$
for k **from** 15 **downto** 0 **do**
  for j **from** $N-1$ **downto** 0 **do**
    $R \leftarrow R + (T[B_{16j+k}] + S[D_{16j+k}])x^{64j}$
  end for
  $R \leftarrow R\cdot x^4$
end for

Table 3. Complexity of the optimizations AB, AC and $AB+CD$ on `CombMul`

Algorithm	#WXOR	#WShift	#WAND
CombMul_ABAC	$2nN + 74N - 7$	$2n + 132N - 7$	$2n$
CombMul_ABplusCD	$2nN + 58N - 14$	$2n + 86N - 14$	$2n$

We also adapt the optimization $AB + CD$ as follows: the addition is performed before the reconstruction of the two products AB and AC, this means that we have only one recursive reconstruction instead of two. This approach is specified in Algorithm 4.

Complexity of KaratRec_ABAC. In the first recursion we have $3N/2$ WXORs for A_0+A_1, B_0+B_1 and C_0+C_1 plus $5N$ WXORS for the reconstructions of R and S. This leads to the following complexity:

$$\begin{cases} \#WXOR(N)=13N/2+3\,\#\text{WXOR}(N/2), \\ \#WXOR(1)=0. \end{cases} \implies \#WXOR(N) = 13N^{\log_2(3)} - 13N$$

$$\begin{cases} \#Mult64(N)=3\#Mult64(N/2), \\ \#WXOR(1)=2. \end{cases} \implies \#Mult64(N) = 2N^{\log_2(3)}.$$

Complexity of KaratRec_ABpCD. In the first recursion we have $2N$ WXORs for the computations $A_0 + A_1$, $B_0 + B_1$, $C_0 + C_1$ and $D_0 + D_1$ plus $5N/2$ WXORs for the reconstruction of R. The complexity for $N = 1$ is equal to $2Mult64$ plus one WXOR. Based on this, we derive the complexity for the `KaratRec_ABpCD` algorithm:

$$\#WXOR(N) = 10N^{\log_2(3)} - 9N, \qquad \#Mult64(N) = 2N^{\log_2(3)}.$$

3.3 Optimizations $AB + CD$ and AB, AC in KaratComb approach

We here consider the `KaratComb` approach for polynomial multiplication. This approach combines the `Karatsuba` and the `CombMul` methods, we can then apply simultaneously the optimization AB, AC (resp. $AB+CD$) described in Subsection 3.1 and 3.2 for the `CombMul` and `KaratRec` algorithms. The resulting algorithms can be easily derived and the complexities of such algorithms is also a direct consequence of the results of Table 3 and of Algorithm 7. Due to lack of space, we skip the details here.

3.4 Complexity comparison and implementation results

Using the complexity results determined in the former subsections, we can compute the complexities of the multiplication algorithms and their optimized AB, AC and $AB+CD$ counter parts for the polynomial

Algorithm 4 KaratRec_ABpCD(A,B,C,D,N)

require: A, B, C and D are polynomials of word size $N = 2^s$ each.
ensure: $R = AB + CD$
if $N = 1$ **then**
Return$(Mul64(A, B) + Mul64(C, D))$
else
// Splitting in two halves of $N/2$ 64bit words.
$A = A_0 + x^{64N/2}A_1,\ B = B_0 + x^{64N/2}B_1,$
$C = C_0 + x^{64N/2}C_1,\ D = D_0 + x^{64N/2}D_1$
// Additions of the halves
$A_2 = A_0 + A_1,\ B_2 = B_0 + B_1$
$C_2 = C_0 + C_1,\ D_2 = D_0 + D_1$
// Recursive multiplications/additions
$R_0 \leftarrow KaratRec\_ABpCD(A_0, B_0, C_0, D_0, N/2)$
$R_1 \leftarrow KaratRec\_ABpCD(A_1, B_1, C_1, D_1, N/2)$
$R_2 \leftarrow KaratRec\_ABpCD(A_2, B_2, C_2, D_2, N/2)$
// Reconstruction
$R \leftarrow R_0 + (R_0 + R_1 + R_2)x^{64N/2} + R_1x^{64N}$
return(R)
end if

Algorithm 5 KaratRec_ABAC(A,B,C,N)

require: A, B and C are polynomials of word size $N = 2^s$ each.
ensure: $R = A \cdot B$ and $S = A \cdot C$
if $N = 1$ **then**
Return$(Mul64(A, B), Mul64(A, C))$
else
// Splitting in two halves of $N/2$ 64bit words.
$A = A_0 + x^{64N/2}A_1,\ B = B_0 + x^{64N/2}B_1,$
$C = C_0 + x^{64N/2}C_1$
// Additions of the halves
$A_2 = A_0 + A_1,\ B_2 = B_0 + B_1,\ C_2 = C_0 + C_1$
// Recursive multiplications
$R_0, S_0 \leftarrow KaratRec\_ABAC(A_0, B_0, C_0, N/2)$
$R_1, S_1 \leftarrow KaratRec\_ABAC(A_1, B_1, C_1, N/2)$
$R_2, S_2 \leftarrow KaratRec\_ABAC(A_2, B_2, C_2, N/2)$
// Reconstruction
$R \leftarrow R_0 + (R_0 + R_1 + R_2)x^{64N/2} + R_1x^{64N}$
$S \leftarrow S_0 + (S_0 + S_1 + S_2)x^{64N/2} + S_1x^{64N}$
return(R, S)
end if

sizes $m = 233, 409$ and 503. We have also implemented these algorithms on the platforms Intel core 2 duo and Intel core i5. The resulting complexities and timings are reported in Table 4 and 5.

Table 4. Complexity/timing results of the `CombMul` and `KaratComb` variants on a core 2 duo

Algorithm	Overall complexity in terms of word operations	503		409		233	
		#W.Op.	μs	#W.Op.	μs	#W.Op.	μs
CombMul	$nN + 2n + 117N - 26$	2182	1.04	1732	0.80	808	0.29
KaratComb	$3nN/2 + n + 219N - 85$	3117	0.77	2582	0.71	1325	0.24
KaratComb_ABAC	$3nN/2 + 6n + 531N - 54$	6462	1.41	5363	1.29	2778	0.44
KaratComb_ABplusCD	$3nN/2 + 9n/2 + 233N - 77$	3866	1.35	3099	1.17	1475	0.42
CombMul_ABAC	$2nN + 4n + 338N - 21$	5203	2.01	4199	1.53	2039	0.57
CombMul_ABplusCD	$2nN + 4n + 144N - 28$	3644	2.01	2834	1.39	1256	0.49

#W. Op. = number of word operations

Table 5. Complexity/timing results of the `KaratRec` variants on a core i5

Algorithm	Complexity for $N = 2^s$		409			233		
	#WXOR	#Mul64	#WXOR	#Mul64	μs	#WXOR	#Mul64	μs
KaratRec	$8N^{\log_2(3)} - 8N$	$N^{\log_2(3)}$	152	27	0.123	40	9	0.079
KaratRec_ABAC	$13N^{\log_2(3)} - 13N$	$2N^{\log_2(3)}$	247	54	0.263	65	18	0,131
KaratRec_ABpCD	$10N^{\log_2(3)} - 9N$	$2N^{\log_2(3)}$	198	54	0.271	54	18	0.166

Based on the results presented above, we remark that the optimization $AB + CD$ has always a better complexity than the optimization AB, AC. Moreover, `KaratRec` approaches seems not interesting in terms of complexities, but timings results on the core 2 duo gives a contradictory conclusion. Furthermore, the implementation results on the core i5 seems to contradict the complexity values. But we might be aware that, due to data dependencies and compiler optimizations, the above implementation

results might be slightly different when used for a specific sequence of operations like the one of scalar multiplication. We also mention that the timings on the core 2 for $m = 233$ is better than the timing reported in [1] for this same field but for core 2 with a smaller frequency.

4 Implementations of scalar multiplication based on the optimizations AB, AC and $AB + CD$

We present in this section our experimental results for scalar multiplication based on the optimizations AB, AC and $AB+CD$ presented in the previous section. We first review best known elliptic curve point operation formulas, and describe how we use the optimizations AB, AC and $AB+CD$ in these formulas. We then present implementation features for two platforms: Intel core 2 duo which does not have any carry less multiplication instruction and Intel core i5 which has such specific instruction.

4.1 Elliptic curve arithmetic

The considered curves are ordinary binary elliptic curve defined by the following Weierstrass equation

$$y^2 + xy = x^3 + x^2 + b \text{ where } b \in \mathbb{F}_{2^m}.$$

We review Lopez-Dahab elliptic curve operations [8], in order to describe how the optimized operations AB, AC and $AB + CD$ can be used in the curve operations. Lopez and Dahab use a specific projective coordinates $P = (X : Y : Z)$ which corresponds to the affine point $(X/Z, Y/Z^2)$. In the following formulas we will use the following notations: $A \cdot B$ is a non reduced polynomial multiplication, and $[R]$ represents the reduction of the polynomial R modulo the irreducible polynomial defining the field $\mathbb{F}_{2^m}$.

Point doubling in Lopez-Dahab coordinates. We compute the doubling $P_1 = (X_1 : Y_1 : Z_1) = 2 \cdot (X : Y : Z)$ of a point $P = (X : Y : Z)$ by performing the following sequence of operations

$$S_X = [X^2], \quad S_Z = [Z^2], \quad S_Y = [Y^2], \quad T_X = [S_X^2], \quad T_Z = [S_Z^2], \quad U = [b \cdot T_Z],$$

and then

$$X_1 = T_X + U, \quad Z_1 = [S_X \cdot S_Z], \quad Y_1 = [\underbrace{U \cdot Z_1 + X_1 \cdot (Z_1 + S_Y + U)}_{AB+CD}].$$

Point addition in Lopez-Dahab coordinates. We review the Lopez-Dahab formula for mixed point addition: we add one point $P_1 = (X_1, Y_1, Z_1)$ which has a regular Lopez-Dahab projective coordinates with a point $P_2 = (X_2, Y_2, 1)$ which is in affine coordinates, i.e., $Z_2 = 1$. The coordinates of $P_3 = (X_3 : Y_3 : Z_3)$ is then computed with the following sequence of operations:

$$S_{Z_1} = [Z_1^2], \quad A = [Y_2 \cdot S_{Z_1}] + Y_1, \quad S_A = A^2, \quad B = [X_2 \cdot Z_1] + X_1, \quad C = [Z_1 \cdot B,], \quad S_B = [B^2],$$

followed by

$$D = S_B \cdot (C + S_{Z_1}), \quad E = [A \cdot C], \quad F = E + Z_3, \quad G = \underbrace{X_2 \cdot E + \overbrace{Y_2 \cdot Z_3}^{AB,AC}}_{AB+CD},$$

and

$$X_3 = [S_A + D] + E, \quad Z_3 = [C^2], \quad Y_3 = \underbrace{X_3 \cdot E + \overbrace{Z_3 \cdot G}^{AB,AC}}_{AB+CD}.$$

In the above formulas, we have indicated the operations which can be performed with the optimization $AB + CD$ and the operations which can be performed with the optimization AB, AC.

Remark 1. Another efficient doubling and mixed-addition formulas are the Kim-Kim formulas [6]. Due to lack of space we do not explain in details how we use the optimizations AB, AC and $AB + CD$ in these formulas.

Scalar multiplication algorithm. The scalar multiplication on the curve $E(\mathbb{F}_{2^m})$ consists of the computation of $r \cdot P$ for a given point $P \in E(\mathbb{F}_{2^m})$ and an m-bit integer r. In our implementations of scalar multiplication, we have recoded r with the NAF_w algorithm [4] and a window size $w = 4$. The scalar multiplication is then performed through the double-and-add algorithm involving a table computation $T[i] = i \cdot P$ for the odd integers $0 < i < 2^{w-1}$. This is a classical approach, the reader may refer to [4] for further details.

4.2 Implementation results on an Intel core 2 duo

On an Intel core 2 duo, there is no carry less multiplication instruction. The available instructions for binary field arithmetic are the classical WShift, WXOR and WAND. We have thus implemented polynomial multiplication using the `CombMul` and `KaratComb` algorithms which do not require any carry less multiplication instruction. For the other field operations we have used the following strategies:

- *Squaring.* Following the strategy described in [4], we use a table Sq which stores the squaring of all 8-bit polynomials. Squaring an N words polynomial is then performed by a sequence of shiftings and maskings to decompose the polynomial as an 8-bit sequence and by accumulating the values output by the table Sq for each 8-bit sub-word.
- *Reduction.* The reduction also follows the strategy of [4]. The fact that the considered irreducible polynomials are sparse (cf. Table 1), this makes possible to perform a reduction with a short sequence of WShifts and WXORs. The reduction is performed only when it is necessary: for example in the curve operations reviewed Subsection 4.1 the reductions are specified with brackets $[\cdot]$. This strategy is generally referred to as the *lazy reduction* strategy [1].
- *Inversion.* The inversion is computed using the Itoh-Tsujii algorithm [5]. This algorithm consists of a sequence of multiplications and multi-squarings. This sequence of multiplication and squaring reconstructs step by step the exponent of $A^{-1} = A^{2^m-2}$ following an addition chain in the exponent. For example, for $m = 233$, the inverse of A is given by $(A^{2^{232}-1})^2$, and this is obtained with the addition chain $1 \to 2 \to 3 \to 6 \to 7 \to 14 \to 28 \to 29 \to 58 \to 116 \to 232$ in the exponent. We have tested two variants of this approach: the first one performs the multi-squaring through a sequence of squaring, and the second variant uses look-up table for large multi-squaring.

The timings of our implementation are reported in Table 6. These values where obtained on a Linux Ubuntu 11.10 platform with GCC 4.6.1. The reported clock-cycles have been obtained with the following strategy: we used the cycle counter `rdtsc` attached to each core in the Intel core 2 duo to get the number of clock cycles. The programs where forced to run on a specific core (we used the software `schedtool` for this) to avoid miss-evaluations due to system scheduling. The reported values are average timings for randomly generated input datas.

Based on the results reported in Table 6, we remark that the proposed optimization $AB + CD$ provides some significant improvements for the field sizes 409 and 503. But the improvement is not really significant for the field size 233. The optimization AB, AC does not provide any significant improvement compared non-optimized implementation. This partially contradicts implementation results of Subsection 3.4 for independent multiplications. This could be explained by some cache conflict. Table 6 also shows that the implementations based on the `KaratComb` algorithm becomes competitive for the field size 503 for regular multiplications.

4.3 Implementation results on an Intel core i5

Intel have implemented a carry less multiplication instruction on the processor Intel core i5. This instruction is really interesting to accelerate binary field arithmetic. Taverne *et al.* have shown in [12] the impact of this new instruction in the performance of software implementation of ECC . We have implemented polynomial multiplication using `KaratRec` algorithm and the carry free multiplication instruction. The other field operations and optimizations used in our implementation are the followings:

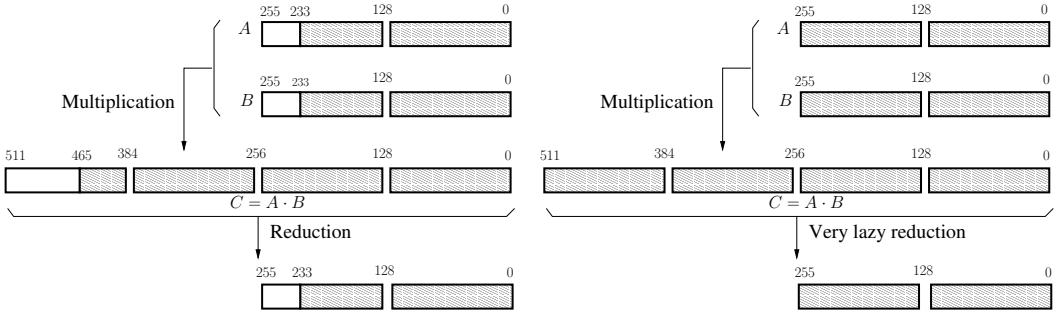
- *128-bit words.* In the Intel core i5 we can use 128-bit registers, called multimedia registers. The pclmul instruction which performs carry free multiplications uses data of this type. We can also perform WXOR and WAND on these 128-bit registers. But the shifts on these registers are slightly more tricky, since there is no regular shift on 128-bit data. Indeed, the shift operates only on the two 64-bit halves.

Table 6. Timings of Scalar multiplication on a 2.5 GHz Intel Core 2 Duo

Optim.	Mul. algorithms	$n = 503$		$n = 409$		$n = 233$	
		#CC	Timing (ms)	#CC	Timing (ms)	#CC	Timing (ms)
none	RegMul with Comb	6925808	2.770	4222807	1.689	**944466**	**0.377**
	RegMul with KaratComb	6437695	2.575	5502652	2.201	968710	0.387
AB,AC	RegMul with Comb AB,AC with Comb	7940830	3.176	4971367	1.988	1334506	0.533
	RegMul with Comb AB,AC with KaratComb	6819583	2.727	4356169	1.742	948857	0.379
	RegMul with KaratComb AB,AC with Comb	6788955	2.715	4998172	1.999	972874	0.389
	RegMul with KaratComb AB,AC with KaratComb	6417570	2.567	4779126	1.911	962438	0.384
AB+CD	RegMul with Comb AB+CD with Comb	6828034	2.731	**4049980**	**1.619**	949781	0.379
	RegMul with Comb AB+CD with KaratComb	6201221	2.480	4054142	1.621	**943446**	**0.377**
	RegMul with KaratComb AB+CD with Comb	6691398	2.676	4364475	1.745	958233	0.383
	RegMul with KaratComb AB+CD with KaratComb	**6056527**	**2.422**	4343174	1.737	952078	0.380

– *Squaring.* The squaring is performed using the pclmul instruction: the N words $A[i], i = 0, \ldots, N-1$ of the polynomial A are squared with N independent pclmul instructions.
– *Reduction.* Following [4], we performed the reduction with a short sequence of shifts and WXORs. We have adapted this approach to deal with the lack of regular shift in 128-bit approach. The code is then slightly larger than a regular 64-bit code, since a regular shift on a 128-bit register requires several instructions.
– *Very lazy reduction.* We have tested an optimization in the case of field size 233. In this case the reduction is done only partially: the polynomial is reduced to a degree 255 instead of 232. Since the `KaratRec` algorithm treats polynomials of degree 233 and 255 in the same way, this *very lazy reduction* is sufficient. Fig. 1 illustrates this strategy: we can see in this diagram that the very lazy reduction save the computations involved in the reduction of the word containing coefficients $c_{255}, \ldots, c_{233}$.

Fig. 1. Regular reduction vs very lazy reduction



(a) Regular reduction (b) Very lazy reduction

– *Inversion.* The inversion is performed using the Itoh-Tsujii algorithm [5]. The best approach here is the version of Itoh-Tsujii which uses look-up tables for multi-squaring. The reported results in Table 7 were obtained using this approach.

We note that, for the field size 233, the very lazy reduction provides a significant improvement compared to regular implementations. We also remark that the optimization $AB + CD$ provides the best result: for the field size 233, this is obtained with Kim-Kim curve operations [6] and for the field

Table 7. Timings of scalar multiplication on an Intel Core i5

Optimization	Curve Formulas	$n = 409$		$n = 233$	
		#CC	Timing (ms)	#CC	Timing (ms)
none	KK	787385	0.314	201312	0.080
	LD	802773	0.321	195203	0.078
Very Lazy Red.	KK	-	-	**177419**	**0.071**
	LD	-	-	184260	0.073
AB, AC and Very Lazy Red.	KK	794071	0.317	178589	0.071
	LD	820696	0.328	185090	0.074
$ABplusCD$ and Very Lazy Red.	KK	778806	0.311	**176852**	**0.070**
	LD	**768711**	**0.307**	183770	0.073

size 409 this was obtained with Lopez-Dahab curve operations. We now compare our results with best known results found in the literature: Taverne *et al.* in [12] report implementation results of scalar multiplication based on Kim-Kim curve operations, NAF_w with $w = 4$ and double-and-add. The results of Taverne *et al.* are of 216,700 clock cycles for $m = 233$ and 871,500 clock cycles for $m = 409$. Our results improve significantly these results.

5 Conclusion

We have presented in this paper software optimizations of binary field operations AB, AC and $AB+CD$ for scalar multiplication on binary elliptic curves. We have established several algorithms for these optimizations and have evaluated the complexity of the corresponding C-like codes of these algorithms. We have then presented implementation results for scalar multiplication on an Intel core 2 duo and on an Intel core i5. In our implementations of scalar multiplication we have used best known algorithms. We have also tested some further optimized operation like the very lazy reduction. The experimental results have shown that for the field sizes 409 and 503 the proposed $AB + CD$ optimization improves significantly the scalar multiplication on an Intel core 2 duo and an Intel core i5. For the case of Intel core i5, we also obtained some significant improvements compared to the best know results found in the literature [12].

References

1. R. Maria Avanzi and N. Thériault. Effects of Optimizations for Software Implementations of Small Binary Field Arithmetic. In *WAIFI 2007*, volume 4547 of *LNCS*, pages 69–84. Springer, 2007.
2. E.R. Berlekamp. Bit-serial Reed-Solomon encoder. *IEEE Transactions on Information Theory*, IT-28, 1982.
3. H. Cohen, A. Miyaji, and T. Ono. Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In *ASIACRYPT*, pages 51–65, 1998.
4. D. Hankerson, J. López Hernandez, and A. Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *LNCS*, pages 1–24. Springer, 2000.
5. 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.
6. Kwang Ho Kim and So In Kim. A New Method for Speeding Up Arithmetic on Elliptic Curves over Binary Fields. Technical report, National Academy of Science, Pyongyang, D.P.R. of Korea, 2007.
7. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.
8. J. López and R. Dahab. Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n). In *Selected Areas in Cryptography, 1998*, pages 201–212, 1998.
9. J. López and R Dahab. High-Speed Software Multiplication in $\mathbb{F}_{2^m}$. In *INDOCRYPT 2000*, volume 1977 of *LNCS*, pages 203–212. Springer, 2000.
10. V. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology, Proceedings of CRYPTO'85*, volume 218 of *LNCS*, pages 417–426. Springer-Verlag, 1986.

11. C. Paar. A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields. In *Brief Contributions*, volume 45 of *IEEE Transactions on Computers*, page 856. IEEE, 1996.
12. J. Taverne, A. Faz-Hernández, D. F. Aranha, F. Rodríguez-Henríquez, D. Hankerson, and J. López. Software Implementation of Binary Elliptic Curves: Impact of the Carry-Less Multiplier on Scalar Multiplication. In *Cryptographic Hardware and Embedded Systems - CHES 2011*, volume 6917 of *LNCS*, pages 108–123. Springer, 2011.

Algorithm 6 CombMul_C_Code

Require: A and B two N words polynomials of nibble length n
Ensure: $R = A \times B$

```
for(i = 0; i < N; i++)                                  }
    T[0][i] = 0;                                        }
    T[1][i] = A[i];                                     }
end for                                                 }
for(k = 2; i < 16; i+= 2)                               }   Table: T[u] = u · A with deg u < 4
    T[k][0] = (T[k >> 1][0] << 1);                      }   #WXOR = 7(2(N − 1) + 1) = 14N − 13
    T[k + 1][0] = T[k][0] ∧ A[0];                       }   #WShift = 7(2(N − 1) + 1) = 14N − 13
    for(i = 1; i < N; i++)                              }
        T[k][i] = (T[k >> 1][i] << 1)                   }
                  ∧(T[k >> 1][i − 1] >> 63);            }
        T[k + 1][i] = T[k][i] ∧ A[i];                   }
    end for                                             }
end for
for(k = 15; k >= n − 16(N − 1); k − −)
    for(j = 0; j < N − 1; j++)
        u = (B[j] >> (4 ∗ k)) & 0xf                     }   Accumulation R ← R + x^{64j} B_{k+16j} A
        for(i = 0; i < N; i++)                          }           #WXOR = N
            R[i + j] = R[i + j] ∧ T[u][i];              }           #WShift = 1
        end for                                         }           #WAND = 1
    end for
    carry = 0
    for(i = 0; i < 2 ∗ N; i++)                          }
        temp = R[i];                                    }   Shift R ← R << 4
        R[i] = (R[i] << 4) ∧ carry;                     }     #WXOR = 2N
        carry = temp >> 60;                             }     #WShift = 4N
    end for                                             }
end for
for(k = n − 16(N − 1) − 1; k > 0; k − −)
    for(j = 0; j < N − 1; j++)
        u = (B[j] >> (4 ∗ k)) & 0xf                     }   Accumulation R ← R + x^{64j} B_{k+16j} A
        for(i = 0; i < N; i++)                          }           #WXOR = N
            R[i + j] = R[i + j] ∧ T[u][i];              }           #WShift = 1
        end for                                         }           #WAND = 1
    end for
    carry = 0
    for(i = 0; i < 2 ∗ N; i++)                          }
        temp = R[i];                                    }   Shift R ← R << 4
        R[i] = (R[i] << 4) ∧ carry;                     }     #WXOR = 2N
        carry = temp >> 60;                             }     #WShift = 4N
    end for                                             }
end for
for(j = 0; j < N; j++)
    u = B[j] & 0xf;                                     }   Accumulation R ← R + x^{64j} B_{16j+k} A
    for i = 0 to N − 1                                  }           #WXOR = N
        R[i + j] = (R[i + j] << 4) ∧ T[u];              }           #WShift = 0
    end for                                             }           #WAND = 1
end for
```

Algorithm 7 `KaratComb_C_code`(A,B)

Require: A and B two polynomials of word size N and nibble size n
Ensure: $R = A \times B$
$s \leftarrow (n/2)$ % 16; // $B_{64(N-1)+s}$ is the most significant nibble of B
$mask \leftarrow (1 << 4 * (s+1)) - 1$;

for$(i = 0; i < N; i++)$
  $A0[i] = A[i]$;
  $B0[i] = A[i]$;
end for
$A0[N/2-1] = A[N/2-1]$ & $mask$;
$B0[N/2-1] = B[N/2-1]$ & $mask$;

Lower halves: A0 and B0
#WAND = 2

for$(i = 0; i < N/2; i++)$
  $A2[i] = (A[i+N/2] << (4*s))$
    $\wedge (A[i+N/2-1] >> (64-4*s))$;
  $B2[i] = (B[i+N/2] << (4*s))$
    $\wedge (B[i+N/2-1] >> (64-4*s))$;
end for

Upper halves: A2 and B2
#WXOR = N
#WShift = 2N

for$(i = 0; i < N/2; i++)$
  $A1[i] = A0[i] \wedge A2[i]$;
  $B0[i] = B0[i] \wedge B2[i]$;
end for

sums: A1 = A0 + A2
and B1 = B0 + B2
#WXOR = N

// The three half size products
$R0 = \texttt{CombMul}(A0, B0)$;
$R1 = \texttt{CombMul}(A1, B1)$;
$R2 = \texttt{CombMul}(A2, B2)$;
// Reconstruction

for$(i = 0; i < N; i++)$
  $R[i] = R0[i]$
end for

$R \leftarrow R0$

$R[N-1] = R[N-1] \wedge (R2[0] << (8*s))$;
for$(i = 1; i < N; i++)$
  $R[i+N] = (R2[i+1] << (8*s)) \wedge (R2[i-1] >> (64-8*s))$;
end for

$R \leftarrow R + R2\, x^{4n}$
#WXOR = N
#WShift = 2N-1

$temp = R0[0] \wedge R1[0] \wedge R2[0]$;
$R[i+N/2-1] = R[i+N/2-1] \wedge (temp << (4*s))$;
for$(i = 0; i < N; i++)$
  $carry = temp << (4*s)$;
  $temp = R0[i] \wedge R1[i] \wedge R2[i]$;
  $R[i+N/2] = R[i+N/2] \wedge (temp >> (64-4*s)) \wedge carry$;
end for

$R \leftarrow R + (R0 + R1 + R2) x^{4(n/2)}$
#WXOR = 4N+3
#WShift = 2N+1

Algorithm 8 CombMul(A,B,C)

Require: Three $64N$-bit binary polynomials $A(x)$, $B(x)$ and $C(x)$ and $B = \sum_{i=0}^{n} \sum_{k=0}^{15} B_{16j+k} x^{4k+64j}$ and $C = \sum_{i=0}^{n} \sum_{k=0}^{15} C_{16j+k} x^{4k+64j}$ are decomposed into nibbles.
Ensure: $R(x) = A(x) \cdot B(x), R'(x) = C(x) \cdot B(x)$
  // Computation of the table T such that $T[u] = u(x) \cdot A(x)$ for all $\deg u(x) < 4$
  $T[0] \leftarrow 0$;
  $T[1] \leftarrow 0$;
  for k **from** 1 **to** 8 **do do**
    $T[2k] \leftarrow T[k] \cdot x$;
    $T[2k+1] \leftarrow T[2k] + A$;
  end for
  // right-to-left shift Comb multiplication AB
  $R \leftarrow 0$
  for k **from** 15 **downto** 0 **do**
    for j **from** $N-1$ **downto** 0 **do**
      $R \leftarrow R + T[B_{16j+k}] \cdot x^{64j}$
    end for
    $R \leftarrow R \cdot x^4$
  end for
  // right-to-left shift Comb multiplication AC
  $R' \leftarrow 0$
  for k **from** 15 **downto** 0 **do**
    for j **from** $N-1$ **downto** 0 **do**
      $R' \leftarrow R' + T[B_{16j+k}] \cdot x^{64j}$
    end for
    $R' \leftarrow R' \cdot x^4$
  end for