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

Christophe Negre[1,2] and Jean-Marc Robert[1,2]

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

Abstract. A scalar multiplication over a binary elliptic curve consists in 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 and Intel Core i5. These experimental results show some significant improvements in the timing of scalar multiplication due to the proposed optimizations.

Keywords: Optimized field operations AB, AC and $AB+CD$, double-and-add, halve-and-add, parallel, scalar multiplication, software implementation, carry-less multiplication.

1 Introduction

Finite field arithmetic is widely used in elliptic curve cryptography (ECC) [13,11] and coding theory [4]. The main operation in ECC is the scalar multiplication which is computed as a sequence of multiplications and additions in the underlying field [6,8]. Efficient implementations of these sequences of finite field operations are thus crucial to get efficient cryptographic protocols.

We 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 in 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 size m polynomials. Specifically, an addition is a simple bitwise XOR of the coefficients: in software, this consists in 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.

A. Youssef, A. Nitaj, A.E. Hassanien (Eds.): AFRICACRYPT 2013, LNCS 7918, pp. 279–296, 2013.

Until the end of 2009 the fastest algorithm for software implementation of polynomial multiplication was the Comb method of Lopez and Dahab [12]. 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 Aranha *et al.* in [1] on an Intel Core 2. But, since the introduction by Intel of a new carry-less multiplication instruction on the new processors i3, i5 and i7, the authors in [16] have shown that the polynomial multiplication based on Karatsuba method [15] outperforms the former approaches based on Lopez-Dahab multiplication. In the sequel, we consider implementations on two platforms: processor without carry-less multiplication (Intel Core 2) and processor i5 which has such instruction.

Our Contributions. In this paper, we investigate 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 [2]. 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 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 also show that this approach can be adapted to the case of Karatsuba multiplication and we evaluate the resulting complexity.

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

Organization of the Paper. In Section 2, we review the best known algorithms for software implementation of polynomial multiplication of size $m \in [160, 600]$. In Section 3, we then present 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 and on an Intel Core i5. Finally, in Section 5, we give some concluding remarks.

2 Review of Multiplication Algorithms

The problem considered in this section is to compute efficiently a multiplication in a binary field $\mathbb{F}_{2^m}$. A field $\mathbb{F}_{2^m}$ can be 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 in multiplying two polynomials of degree at most $m-1$ and reducing the product modulo $f(x)$. The fields 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 [8] for further details).

We then 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 of 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 $m = 233$ and $m = 409$ considered in this paper.

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

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

2.1 Comb Multiplication

One of the best known methods for software implementation of the multiplication of two polynomials A and B was proposed by Lopez and Dahab in [12]. 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, based on our experiments and several other experimental results in the literature [1,8], this seems to be the best case for the platform considered here (Intel Core 2). 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

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

where $\deg B_{16j+k} < 4$. Then the product $R = A \times B$ is expressed by expanding the above expression of B as follows

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

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 formulation and Algorithm 6 in the appendix for a C-like code formulation.

Complexity. We evaluate the complexity of the corresponding C-like code (Algorithm 6; see p. 294) of the `CombMul` algorithm in terms of the number

Algorithm 1. `CombMul`(A,B)

Require: Two binary polynomials $A(x)$ and $B(x)$ of degree $< 64N - 4$, and $B(x) = \sum_{j=0}^{N-1}\sum_{k=0}^{15} B_{16j+k}x^{4k+64j}$ is decomposed in 64-bit 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 $\deg u(x) < 4$
$T[0] \leftarrow 0$;
$T[1] \leftarrow A$;
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**
  $R \leftarrow R \cdot x^4$
  for j **from** $N-1$ **downto** 0 **do**
    $R \leftarrow R + T[B_{16j+k}]x^{64j}$
  end for
end for

of 64-bit word operations (WXOR, WAND and WShift). We do not count the operations performed for the loop variables $k, j, \ldots$. Indeed, when all the loops are 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 nested loops: the one on k is of length 15 and the loop on i is of length $2N$. The loop operations consist in 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 (except for $k = 0$).

As stated in Table 2, the total number of operations is equal to $nN + 44N - 7$ WXORs, $n + 73N - 7$ WShifts and n WANDs.

2.2 Karatsuba Multiplication

We review the Karatsuba approach for binary polynomial multiplication. Let A and B be two binary polynomials of size $64N$ and assume that N is even. Then, we first split A and B in two halves $A = A_0 + x^{64N/2}A_1$ and $B = B_0 +$

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

$x^{64N/2}B_1$ and then we re-express the product $A \times B$ in terms of three polynomial multiplications of half size:

$$\begin{array}{l} R_0 = A_0B_0, \quad R_1 = A_1B_1, \quad R_2 = (A_0 + A_1)(B_0 + B_1), \\ C = R_0 + x^{64N/2}(R_0 + R_1 + R_2) + x^{64N}R_1. \end{array} \tag{1}$$

The resulting recursive approach is given in `KaratRec` algorithm (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 a `Mult64` operation. 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,N)

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 halves of word size $N/2$.
    $A = A_0 + x^{64N/2}A_1$
    $B = B_0 + x^{64N/2}B_1$
    // Recursive multiplication
    $R_0 \leftarrow \texttt{KaratRec}(A_0, B_0, N/2)$
    $R_1 \leftarrow \texttt{KaratRec}(A_1, B_1, N/2)$
    $R_2 \leftarrow \texttt{KaratRec}(A_0 + A_1, B_0 + B_1, N/2)$
    // 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).

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

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

In this section, we present 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 mentionned for example in [2] 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.

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 $R\leftarrow R+(T[B_{16j+k}]+S[D_{16j+k}])x^{64j}$ 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.

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
  $R \leftarrow 0$
  for k **from** 15 **downto** 0 **do**
    $R \leftarrow R \cdot x^4$
    for j **from** $N - 1$ **downto** 0 **do**
      $R \leftarrow R + (T[B_{16j+k}] + S[D_{16j+k}])x^{64j}$
    end for
    return (R)
  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	$2nN + 58N - 14$	$2n + 86N - 14$	$2n$

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.

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:

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 $N/2$ 64-bit 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 $N/2$ 64-bit 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

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

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

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 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 sizes $m = 233$ and $m = 409$. We implemented these algorithms on the platforms Intel Core 2 and Intel Core i5. Our implementation uses 128-bit registers and vector instructions available on these two processors. On the Core 2 we used the modified `CombMul` algorithm of [5,1] which uses mostly shifts by multiple of 8; cheaper than an arbitrary shift for 128-bit data. On the Core i5 we implemented the `KaratRec` multiplication method with the PCLMUL instruction which performs carry-less multiplication of two 64 bit inputs contained in 128-bit registers.

The resulting complexities and timings are reported in Table 4 and Table 5.

Table 4. Complexity/timing results of the `CombMul` variants on a Core 2 (2.5 GHz)

Algorithm	Overall complexity in terms of word operations	233		409	
		#W.Op.	#CC	#W.Op.	#CC
CombMul	$nN + 2n + 117N - 14$	808	336	1732	795
CombMul_ABAC	$2nN + 4n + 206N - 14$	1511	555	3282	1597
CombMul_ABplusCD	$2nN + 4n + 144N - 28$	1256	564	2834	1737

#W. Op. = number of word operations (WXOR, WAND, WShift).
#CC = number of clock cycles.

Table 5. Complexity/timing results of the `KaratRec` variants on a Core i5 (2.5 GHz)

Algorithm	Complexity for $N = 2^s$		233			409		
	#WXOR	#Mul64	#WXOR	#Mul64	#CC	#WXOR	#Mul64	#CC
`KaratRec`	$8N^{\log_2(3)} - 8N$	$N^{\log_2(3)}$	40	9	107	152	27	286
`KaratRec_ABAC`	$13N^{\log_2(3)} - 13N$	$2N^{\log_2(3)}$	65	18	189	247	54	566
`KaratRec_ABpCD`	$10N^{\log_2(3)} - 9N$	$2N^{\log_2(3)}$	54	18	182	198	54	541

Based on the results presented above, we notice that the optimization $AB + CD$ has always a better complexity than the optimization AB, AC and better than two independent multiplications. Concerning the timings we note that:

- On the Core 2 the optimization *ABplusCD* is always slower than the optimization AB, AC. Moreover, the optimizations *ABplusCD* and AB, AC are effective only for $m = 233$, since in this case they are faster than two independent multiplications. This seems to contradict the corresponding complexity results since the complexity differences appear quite large.
- On the Core i5 the timing results are more related to the complexity values: for the two considered degrees *ABplusCD* and AB, AC are faster than two independent multiplications and *ABplusCD* is always faster than AB, AC.

In the literature we can find some timing of the `CombMul` algorithm over a Core 2 in [1]. The authors in [1] report implementation timings in the range of $[241, 276]$ clock-cycles for a polynomial multiplication of size $m = 233$ and in the range of $[690, 751]$ for $m = 409$, which are both better than the results reported in Table 4. Our results on the Core i5 compares favorably with the results reported in [16]: 128 clock-cycles for $m = 233$ and 345 clock-cycles for $m = 409$. These reported timings may include the reduction operation (this is not clearly specified in [16]). The same authors reported later in [17] better timings on the same processor and compiler: 100 clock-cycles for $m = 233$ and 270 clock-cycles for $m = 409$.

4 Implementations of Scalar Multiplication Based on the Optimizations AB, AC and $AB + CD$

In this section, we present 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. Then we describe the strategies we used for our implementations: scalar multiplication algorithms and implementations of field operations (squaring, inversion, ...). Finally, we present the implementation results on an Intel Core 2 and an Intel Core i5.

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 will more specifically focused on the two NIST [14] curves $B233$ and $B409$.

Optimization *AB*, *AC* and *ABplusCD* in Curve Operation. We review Kim-Kim elliptic curve operations [10] in order to describe how the optimized operations AB, AC and $AB + CD$ can be used in the curve operations. Kim and Kim in [10] use a specific projective coordinates $P = (X : Y : Z : T)$ which corresponds to the affine point $(X/Z, Y/T)$ where $T = Z^2$. In the following formulas we 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 Kim-Kim coordinates.* We compute the doubling $P_1 = (X_1 : Y_1 : Z_1 : T_1) = 2 \cdot (X : Y : Z : T)$ of a point $P = (X : Y : Z : T)$ by performing the following sequence of operations

$$A{=}X^2,\ \ B{=}[Y]^2.$$

and then:

$$Z_1{=}[T \cdot A],\ \ T_1{=}[Z_1^2],\ \ X_1{=}[A^2 + \underbrace{b \cdot T^2}_{AB,AC}],\ \ Y_1{=}\overbrace{B \cdot (B + X_1 + Z_1) + \underbrace{b \cdot T_1}_{AB,AC}}^{ABplusCD} + T_1.$$

• *Point addition in Kim-Kim coordinates.* We review the Kim-Kim formula for mixed point addition: we add one point $P_1 = (X_1 : Y_1 : Z_1 : T_1)$ which has a regular Kim-Kim projective coordinates with a point $P_2 = (X_2 : Y_2 : 1 : 1)$ which is in affine coordinates, i.e., $Z_2 = T_2 = 1$. The coordinates of $P_3 = (X_3 : Y_3 : Z_3 : T_3)$ is then computed with the following sequence of operations:

$$A{=}X_1 + [X_2 \cdot Z_1],\ B{=}[Y_1 + Y_2 \cdot T_1],\ C{=}[A \cdot Z_1],\ D{=}\underbrace{[C \cdot (B + C)]}_{AB,AC}.$$

and then deduce $Z_3 = [C^2],\ T_3 = [Z_3^2],$ and

$$X_3{=}[B^2 + \underbrace{C \cdot [A^2]}_{AB,AC}] + D,\ \ Y_3{=}\overbrace{[(X_3 + [X_2 \cdot Z_3]) \cdot D + (X_2 + Y_2) \cdot T_3]}^{ABplusCD}.$$

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

• *Optimization AB, AC and $ABplusCD$ in other curve operation formulas.* We consider the following two cases: Lopez-Dahab formulas, which are variants of the Kim-Kim formulas, and Montgomery laddering. For the Lopez-Dahab formulas the optimizations AB, AC and $ABplusCD$ can be applied in both doubling and mixed addition. For the Montgomery laddering we can just apply one optimization $ABplusCD$ in the inner loop operation.

Another interesting operation is the point halving, but, unfortunately, we could not apply any of the optimizations AB, AC or $ABplusCD$ in the halving formula of [8] (Algorithm 3.81 [8], p. 131]). Indeed, this point halving consists in one half-trace operation, followed by one multiplication, one trace computation and one square root, so no optimization based on combined multiplications can be applied.

Scalar Multiplication Algorithm. The scalar multiplication on the curve $E(\mathbb{F}_{2^m})$ consists in the computation of $r \cdot P$ for a given point $P \in E(\mathbb{F}_{2^m})$ and an ℓ-bit integer r where ℓ is the bit length of the order of P. We implemented the following methods for scalar multiplication:

– *Double-and-add.* This approach consists in a sequence of doublings and additions on the curve. The integer r is generally recoded with the NAF_w algorithm [8] with window size $w = 4$ in order to reduce the number of additions performed during the double-and-add algorithm. The scalar multiplication then requires a table precomputation $T[i] = i \cdot P$ for the odd integers $0 < i < 2^{w-1}$. In our implementations we used the Kim-Kim (cf. Subsection 4.1) and the Lopez-Dahab [8] doubling and addition formulas.
– *Halve-and-add.* This approach consists in a sequence of halvings and additions on the curve. The integer r is first recoded in $r' = r \cdot 2^{\ell-1} \mod \#{<}P{>}$ since in this case we have:

$$r = r' 2^{-(\ell-1)} = (\sum_{i=0}^{\ell-1} r'_i 2^i) 2^{-(\ell-1)} = (\sum_{i=0}^{\ell-1} r'_i 2^{i-(\ell-1)})$$

and we can then compute $r \cdot P$ as a sequence of halvings and additions. We use again the NAF_w algorithm for $w = 4$ to recode r' and the variant of the halve-and-add approach to perform the scalar multiplication. The reader may refer to Section 3.6 in [8] for further details on point halving approaches.

- *Parallel (Double-and-add, Halve-and-add).* This approach, proposed in [16,17], splits the computation of the scalar multiplication in two parts: one uses double-and-add approach and the other uses halve-and-add approach. This requires some recoding of the scalar r similar to the one used in halve-and-add approach.
- *Montgomery.* The last approach we considered is the Montgomery laddering (cf. Algorithm 3.40, p.103 in [8]): it is a variant of the double-and-add approach. The main difference is that two points are computed in the inner for loop of the algorithm: P_1 and P_2 which have a constant difference $P_1 - P_2 = P$. This approach has some nice properties as counter measure against side channel attacks.

4.2 Implementation Aspects

We use the following strategies to implement the field operations required in scalar multiplication algorithms:

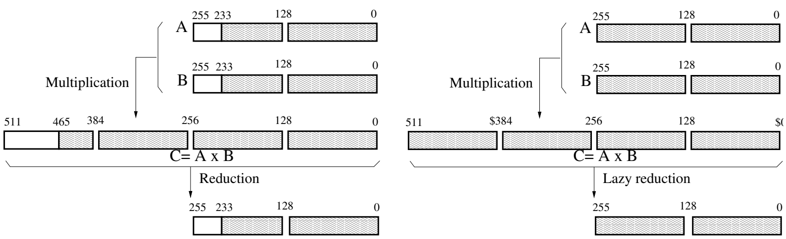
- *Multiplication.* The considered multiplication strategies have already been described in Subsection 3.3. Specifically, on the Intel Core 2 platform, we use the version of the `CombMul` algorithm of [5,1] which uses 128-bit instruction sets. On the Intel Core i5 platform we use the Karatsuba algorithm along with vector instructions and more precisely the carry-less instruction which performs binary polynomial multiplication of size 64 bits.
- *Squaring.* For the squaring we use the strategy described in [1]. Specifically, we use a 128-bit word Sq which stores in each byte the squaring of a 4-bit polynomial. Then for each 128-bit word $A[i]$ of A we separate odd and even nibbles with a masking and a shift and then apply `_mm_shuffle_epi8` intrinsinc function with left input value Sq and right input value the word containing even or odd nibbles of $A[i]$. The result is a 128-bit word containing the squaring of each nibble. The bytes are then reordered and repacked into two 128-bit words. The reader may refer to Algorithm 1 in [1] for further details.
- *Square-root.* The square root is based on the expression $\sqrt{A} = (\sum_{i=0}^{\lceil m/2 \rceil} a_{2i}X^i) + \sqrt{x}(\sum_{i=0}^{\lceil m/2 \rceil} a_{2i+1}X^i)$. Following [1], we separate odd and even coefficients of A using the intrinsinc function named `_mm_shuffle_epi8` and by reordering the resulting bytes. Then the multiplication by $\sqrt{x}$ is done through a number of shifts and additions since for $m = 233$ and $m = 409$, $\sqrt{x}$ has a sparse expression.
- *Reduction.* The reduction follows the strategy of [8]: the considered irreducible polynomials are sparse (cf. Table 1), this makes possible to perform a reduction with a short sequence of shifts and WXORs.
- *Inversion.* The inversion is computed using the Itoh-Tsujii algorithm [9]. This algorithm consists in 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 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. For multi-squaring consisting in long sequence of squaring we use a look-up table approach.

– *Half-trace.* In the halving curve operation, we have to compute half-trace (HT) of an element: $HT(A) = \sum_{i=0}^{(m-1)/2} A^{2^{2i}}$. Our implementation is again inspired from [16] and [7] and uses the intrinsic function `_mm_shuffle_epi8` to compute the half-trace of the even bits of A and look-up table to compute the half-trace of the odd bits of A. For further details on this the reader may refer to [16,17].

Lazy Reductions. An optimization called *lazy-reduction* can be used to optimize curve operations (cf. [2,3]). This consists in removing unnecessary reduction operations performed during the sequence of multiplications and squarings in the curve operation formulas. Here we considered the following two lazy reduction optimizations:

– *Lazy-reduction 1 (LR1).* This optimization regroups reduction operations corresponding to distinct squarings or multiplications. For example in the sequence of operations $A^2 + C \cdot D$ we can perform the addition (addition of polynomial of degree $2m - 2$) before performing the reduction. This reduces the total number of WXORs and WShifts. In the considered elliptic curve operation formulas (Kim-Kim, Lopez-Dahab and Montgomery) the bracket $[\cdot]$ specifies the reduction operations corresponding to this LR1 optimization (cf. Subsection 4.1).
– *Lazy-reduction 2 (LR2).* In this case the reduction modulo the irreducible polynomial is partially done, this results in a polynomial with a degree larger than $m - 1$. We have applied this approach for $m = 233$: the polynomial is reduced to a degree 255 instead of 232. Since the `KaratRec` algorithm multiplies polynomials of size 256, we don't have to reduce the coefficients in the range $[233, 256]$, so we can use a *lazy reduction* of this kind. Figure 4.2 illustrates this strategy: we can see in this figure that the LR2 approach saves the computations involved in the reduction of the word containing coefficients $c_{255}, \ldots, c_{233}$.



(a) Regular reduction (b) Lazy reduction 2

Fig. 1. Regular reduction vs lazy reduction 2

We did not apply this strategy in the case of Intel Core 2 since the `CombMul` approach multiplies polynomials of degree 232 and not 256. For the case of degree 409, the LR2 approach does not provide any saving in the number of words which have to be reduced, so, again, we did not implement such LR2 optimization.

4.3 Implementation Results on an Intel Core 2

The timings of our implementation are reported in Table 6. These values were obtained on a Linux Ubuntu 11.10 platform with GCC 4.6.1. The reported clock-cycles were obtained with the following strategy: we used the cycle counter `rdtsc` attached to each core in the Intel Core 2 to get the number of clock cycles. The reported values are average timings for randomly generated input datas.

Table 6. Timings in terms of 10^3 clock-cycles of scalar multiplication on an Intel Core 2 (2.50GHz)

	Optimization	Formulas	$m = 233$ $(\#CC)/10^3$	$m = 409$ $(\#CC)/10^3$
Double-and-add	none	KK	592	2125
		LD	613	2192
	LR1	KK	1249	2207
		LD	1179	2832
	AB, AC	KK	558	6217
		LD	928	2917
	$ABplusCD$	KK	542	2187
		LD	553	2296
Halve-and-add	none	KK	387	1504
		LD	403	1575
	LR1	KK	651	1706
		LD	855	1837
	AB, AC	KK	858	2277
		LD	887	2359
	$ABplusCD$	KK	375	1504
		LD	386	1640
Parallel$^{(*)}$ (Double-and-add + Halve-and-add)	none	KK	280	965
		LD	295	999
	LR1	KK	335	1042
		LD	315	1104
	AB, AC	KK	270	2311
		LD	289	1362
	$ABplusCD$	KK	273	977
		LD	277	1014
Montgomery	none	-	593	2190
	LR1	-	637	2482
	$ABplusCD$	-	549	2289

(∗) The optimizations AB, AC and $ABplusCD$ are applied only on the double-and-add part.

The experimental results of the lazy-reduction optimization (LR1) do not show the expected speed-ups: all the codes involving such lazy-reduction are all slower than the same code running without it. Consequently, we have not combined this optimization with the two other optimizations AB, AC and $ABplusCD$.

Table 7. Timings in terms of 10^3 clock-cycles of scalar multiplication on an Intel Core i5 (2.5 GHz)

	Optimizations	Curve Formulas	$m = 233$ $\#CC/10^3$	$m = 409$ $\#CC/10^3$
Double-and-add	none	KK	246	917
		LD	252	940
	LR1 and LR2$^{(**)}$	KK	220	906
		LD	228	959
	AB, AC and LR1 and LR2$^{(**)}$	KK	219	903
		LD	226	961
	$ABplusCD$ and LR1 and LR2$^{(**)}$	KK	214	877
		LD	222	903
Halve-and-add	none	KK	165	667
		LD	169	719
	LR1 and LR2$^{(**)}$	KK	150	723
		LD	155	708
	AB, AC and LR1 and LR2$^{(**)}$	KK	149	733
		LD	155	720
	$ABplusCD$ and LR1 and LR2$^{(**)}$	KK	150	696
		LD	154	689
Parallel$^{(*)}$	none	KK	131	466
		LD	133	478
	LR1 and LR2$^{(**)}$	KK	116	458
		LD	122	474
	AB, AC and LR1 and LR2$^{(**)}$	KK	117	457
		LD	123	476
	$ABplusCD$ and LR1 and LR2$^{(**)}$	KK	117	452
		LD	118	467
Montgomery	none	-	244	924
	LR1 and LR2$^{(**)}$	-	229	886
	$ABplusCD$ and LR1 and LR2$^{(**)}$	- -	220	883

(*) The optimizations AB, AC and $ABplusCD$ are applied only on the double-and-add part.
(**) The optimizations LR2 is applied only for $m = 233$

Based on the results reported in Table 6, we remark that the proposed optimization $AB + CD$ provides some significant speed-up for the field sizes 233 only. The optimization AB, AC does also provide some speed-up compared to non-optimized results in the case of $m = 233$, but in some cases we obtain some sudden loose of performance like in halve-and-add or double-and-add/LD cases. In the case $m = 409$, none of the optimizations provide any improvement, this confirms the timings we get in Table 4.

We could not find in the literature any timing on a Core 2 for the same curves and same fields. We just mention that Aranha *et al.* in [1] report in the range [785000,858000] clock-cycles over the curve NIST-B283 and [4310000,4754000] clock-cycles over the curve NIST-B571 for double-and-add scalar multiplication on an Intel Core 2. This means that our timings seem to be in the expected range of values.

4.4 Implementation Results on an Intel Core i5

In Table 7 we report our timings obtained on an Intel Core i5 using implementation strategies discussed in Subsections 4.1 and 4.2. The codes were compiled

Algorithm 6. `CombMul_C_Code`

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

with GCC 4.7.2 on a Linux Ubuntu 12.10. We also disabled the turbo mode of the Core i5 in order to avoid miss-evaluations on the timings.

We note that, the lazy reduction optimizations provide a significant speed-up compared to regular implementations. We also remark that, except in some rare cases, the optimizations $AB + CD$ and AB, AC provide a speed-up compared to non-optimized or LR-optimized implementations. In the case of halve-and-add, the speed-up is less than in the case of double-and-add, but this can be explained by the fact that, in halve-and-add approach, the optimizations are only used in the curve additions which are less frequent than the point halvings. Moreover, the optimization $AB + CD$ is generally more efficient than AB, AC. The only cases in which neither $AB + CD$ nor AB, AC provide the best timing result is the parallel implementation for $m = 233$ and halve-and-add implementation for $m = 409$.

Let us briefly compare our results with the ones obtained by Aranha *et al.* over an Intel Core i5 with a GCC compiler in [17]. We remark that, except for parallel implementation when $m = 409$, our results are competitive with the timings of [17]. This means that our implementations reach the level of performance of [17] and that the proposed optimized operations are efficient when included in the best known implementation strategies for Intel Core i5.

5 Conclusion

The goal of this paper was to study 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 and on an Intel Core i5. In our implementations of scalar multiplication we have used best known algorithms. We have also tested lazy reduction optimizations. The experimental results have shown that the proposed $AB + CD$ optimization improves the timing of scalar multiplication on an Intel Core 2 only for the small field $\mathbb{F}_{2^{233}}$. On an Intel Core i5, the optimization provides the best results for scalar multiplication over the two considered fields $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{409}}$. For the case of Intel Core i5, we have reached the level of performance of the best known results found in the literature [16].

Acknowledgment. We are greatful to the members of the team DALI (University of Perpignan) for their helpful comments on the preliminary versions of this work. This work was supported by the ANR Pavois.

References

1. Aranha, D.F., López, J., Hankerson, D.: Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 144–161. Springer, Heidelberg (2010)
2. Avanzi, R.M., Thériault, N.: Effects of Optimizations for Software Implementations of Small Binary Field Arithmetic. In: Carlet, C., Sunar, B. (eds.) WAIFI 2007. LNCS, vol. 4547, pp. 69–84. Springer, Heidelberg (2007)
3. Avanzi, R.M., Thériault, N., Wang, Z.: Rethinking low genus hyperelliptic Jacobian arithmetic over binary fields: interplay of field arithmetic and explicit formulæ. J. Mathematical Cryptology 2(3), 227–255 (2008)
4. Berlekamp, E.R.: Bit-serial Reed-Solomon encoder. IEEE Trans. on Inform. Theory IT-28 (1982)
5. Beuchat, J.-L., López-Trejo, E., Martínez-Ramos, L., Mitsunari, S., Rodríguez-Henríquez, F.: Multi-core Implementation of the Tate Pairing over Supersingular Elliptic Curves. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 413–432. Springer, Heidelberg (2009)

6. Cohen, H., Miyaji, A., Ono, T.: Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 51–65. Springer, Heidelberg (1998)
7. Fong, K., Hankerson, D., López, J., Menezes, A.: Field Inversion and Point Halving Revisited. IEEE Trans. Computers 53(8), 1047–1059 (2004)
8. Hankerson, D., Menezes, A., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag New York, Inc., Secaucus (2003)
9. Itoh, T., Tsujii, S.: A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. Information and Computation 78, 171–177 (1988)
10. Kim, K.H., Kim, S.I.: 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)
11. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48, 203–209 (1987)
12. López, J., Dahab, R.: High-Speed Software Multiplication in $\mathbb{F}_{2^m}$. In: Roy, B., Okamoto, E. (eds.) INDOCRYPT 2000. LNCS, vol. 1977, pp. 203–212. Springer, Heidelberg (2000)
13. Miller, V.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
14. National Institute of Standards and Technology (NIST). Recommended elliptic curves for federal government use. NIST Special Publication (July 1999)
15. Paar, C.: A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields. IEEE Trans. on Comp. 45, 856 (1996)
16. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Software Implementation of Binary Elliptic Curves: Impact of the Carry-Less Multiplier on Scalar Multiplication. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 108–123. Springer, Heidelberg (2011)
17. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. J. Cryptographic Engineering 1(3), 187–199 (2011)