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

Christophe Negre, Jean-Marc Robert

To cite this version:

HAL Id: hal-00724785
https://inria.hal.science/hal-00724785v2
Submitted on 6 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Impact of Optimized Operations $AB$, $AC$ and $AB + CD$ in Scalar Multiplication over Binary Elliptic Curve

C. Negre$^{1,2}$ and J.-M. Robert$^{1,2}$

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

September 6, 2013

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 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.

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 $A$:

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

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.

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

<table>
<thead>
<tr>
<th>$m$ the field degree</th>
<th>Irreducible polynomial</th>
<th>$N$ (64-bit word size)</th>
<th>$n$ (nibble size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>233</td>
<td>$x^{33} + x^4 + 1$</td>
<td>4</td>
<td>59</td>
</tr>
<tr>
<td>409</td>
<td>$x^{409} + x^{41} + 1$</td>
<td>7</td>
<td>103</td>
</tr>
</tbody>
</table>

The above expression can be computed through a sequence of accumulations $R \leftarrow R + T[B_{16j+k}]x^{64j}$, corresponding to the terms $A \cdot B_{16j+k}x^{64j}$, 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.

Algorithm 1 CombMul$(A,B)$

Require: Two binary polynomials $A(x)$ and $B(x)$ of degree $< 64N - 4$, and $B(x) = \sum_{j=0}^{N-4} \sum_{k=0}^{6} B_{16j+k}x^{64j+k}$ 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

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 \( k, j, \ldots \). 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>#WXOR</th>
<th>#WShift</th>
<th>#WAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table ( T )</td>
<td>( 14N - 7 )</td>
<td>( 14N - 7 )</td>
<td>0</td>
</tr>
<tr>
<td>( R \leftarrow R + T[B_{16j+k}]x^{64j} )</td>
<td>( nN )</td>
<td>( n - N )</td>
<td>( n )</td>
</tr>
<tr>
<td>Shift ( R \leftarrow R &lt;&lt; 4 )</td>
<td>( 30N )</td>
<td>( 60N )</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>( nN + 44N - 7 )</td>
<td>( n + 73N - 7 )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

### 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 \( B = B_0 + x^{64N/2}B_1 \) and then we re-express the product \( AB \) in terms of three polynomial multiplications of half size:

\[
\begin{align*}
R_0 &= A_0B_0, \\
R_1 &= A_1B_1, \\
R_2 &= (A_0 + A_1)(B_0 + B_1), \\
C &= R_0 + x^{64N/2}(R_0 + R_1 + R_2) + x^{64N}R_1.
\end{align*}
\]

The resulting recursive approach is given in the KaratRec algorithm (i.e., 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.

**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.
Algorithm 2 KaratRec(A,B,N)

Require: A and B on N = 2^s computer words.
Ensure: R = A × 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 ← KaratRec(A_0, B_0, N/2)
  R_1 ← KaratRec(A_1, B_1, N/2)
  R_2 ← KaratRec(A_0 + A_1, B_0 + B_1, N/2)
  // Reconstruction
  R ← R_0 + (R_0 + R_1 + R_2)x^{64N/2} + R_1x^{64N}
  return (R)
end if

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 mentioned 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 · 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. Due to lack of space we do not provide the algorithm form of CombMul_ABAC, but it can be easily deduced from Algorithm 1.

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 · A and S[u] = u · C for u of degree < 4
unchanged. But we accumulate $T[B_{i6j+k}]$ and $S[D_{i6j+k}]$ in the same variable $R \leftarrow R + (T[B_{i6j+k}] + S[B_{i6j+k}])x^{64j}$. The shifts by 4 are then performed only on $R$.

Algorithm 3 CombMul_APlusCD(A,B)

Require: Four binary polynomials $A$, $B$, $C$ and $D$ of degree $< 64N - 4$, and $B(x) = \sum_{j=0}^{N-1} \sum_{k=0}^{15} B_{16j+k}x^{1k+64j}$ with deg $B_{16j+k} < 4$ and $D(x) = \sum_{j=0}^{N-1} \sum_{k=0}^{15} D_{16j+k}x^{1k+64j}$ 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
    $R \leftarrow R + (T[B_{i6j+k}] + S[D_{i6j+k}])x^{64j}$
  end for
return $(R)$
end for

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

- We have in the CombMul_APlusCD algorithm two table computations which contribute to twice the complexity of the table computation in Table 2.
- The accumulations $R \leftarrow R + (T[B_{i6j+k}] + S[D_{i6j+k}])x^{64j}$ 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.

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#WXOR</th>
<th>#WShift</th>
<th>#WAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>CombMul_ABAC</td>
<td>$2nN + 74N - 7$</td>
<td>$2n + 132N - 7$</td>
<td>$2n$</td>
</tr>
<tr>
<td>CombMul_APlusCD</td>
<td>$2nN + 58N - 14$</td>
<td>$2n + 86N - 14$</td>
<td>$2n$</td>
</tr>
</tbody>
</table>

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.

### 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** $N = 1$

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

$R_0 \leftarrow \text{KaratRec_ABpCD}(A_0, B_0, C_0, D_0, N/2)$

$R_1 \leftarrow \text{KaratRec_ABpCD}(A_1, B_1, C_1, D_1, N/2)$

$R_2 \leftarrow \text{KaratRec_ABpCD}(A_2, B_2, C_2, D_2, N/2)$

// 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** $N = 1$

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

$R_0, S_0 \leftarrow \text{KaratRec_ABAC}(A_0, B_0, C_0, N/2)$

$R_1, S_1 \leftarrow \text{KaratRec_ABAC}(A_1, B_1, C_1, N/2)$

$R_2, S_2 \leftarrow \text{KaratRec_ABAC}(A_2, B_2, C_2, N/2)$

// 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**

### 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:

$$\#	ext{WXOR}(N) = 13N/2 + 3 \#	ext{WXOR}(N/2), \quad \#	ext{WXOR}(1) = 0$$

$$\#	ext{Mul64}(N) = 3\#	ext{Mul64}(N/2), \quad \#	ext{Mul64}(1) = 2$$

which results in:

$$\#	ext{WXOR}(N) = 13N \cdot \log_2(3) - 13N$$

$$\#	ext{Mul64}(N) = 2N \cdot \log_2(3).$$

### 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 $2\cdot\text{Mul64}$ plus one WXOR. Based on this, we derive the complexity for the KaratRec_ABpCD algorithm:

$$\#	ext{WXOR}(N) = 10N \cdot \log_2(3) - 9N$$

$$\#	ext{Mul64}(N) = 2N \cdot \log_2(3).$$

### 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 \texttt{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)**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Overall complexity in terms of word operations</th>
<th>$2^{233}$</th>
<th>$2^{409}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#W. Op.</td>
<td>#CC</td>
<td>#W. Op.</td>
</tr>
<tr>
<td>CombMul</td>
<td>$8N + 2n + 117N - 14$</td>
<td>868</td>
<td>446</td>
</tr>
<tr>
<td>CombMul ABAC</td>
<td>$2nN + 4n + 206N - 14$</td>
<td>1511</td>
<td>555</td>
</tr>
<tr>
<td>CombMul ABplusCD</td>
<td>$2nN + 4n + 144N - 28$</td>
<td>1256</td>
<td>564</td>
</tr>
</tbody>
</table>

#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)**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity for $N = 2^i$</th>
<th>$2^{233}$</th>
<th>$2^{409}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#WXOR</td>
<td>#Mul64</td>
<td>#WXOR</td>
</tr>
<tr>
<td>KaratRec</td>
<td>$8N^{log_2(3)} - 8N$</td>
<td>40</td>
<td>9</td>
</tr>
<tr>
<td>KaratRec ABAC</td>
<td>$13N^{log_2(3)} - 13N$</td>
<td>65</td>
<td>18</td>
</tr>
<tr>
<td>KaratRec ABpCD</td>
<td>$10N^{log_2(3)} - 9N$</td>
<td>54</td>
<td>18</td>
</tr>
</tbody>
</table>

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. However, our results on the Core i5 compares favorably with the results recently reported in [16], which give 128 clock-cycles for $m = 233$ and 345 clock-cycles for $m = 409$, but these reported timings may include the reduction operation (this is not clearly specified in [16]).

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
\[ y^2 + xy = x^3 + x^2 + b \] where \( b \in F_{2^m} \).

We will more specifically focused on the two NIST [14] curves \( B_{233} \) and \( B_{409} \).

4.1.1 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 \( F_{2^m} \).

- **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
  \[ A = X^2, \quad B = [Y]^2. \]
  and then:
  \[ Z_1 = [T \cdot A], \quad T_1 = [Z_1^2], \quad X_1 = [A^2 + b \cdot T_2^2], \quad Y_1 = B \cdot (B + X_1 + Z_1) + b \cdot T_1 + T_1. \]

- **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:
  \[ A = X_1 + [X_2 \cdot Z_1], \quad B = [Y_1 + Y_2 \cdot T_1], \quad C = [A \cdot Z_1], \quad D = [C \cdot (B + C)] \]
  and then deduce:
  \[ Z_3 = [C^2], \quad T_3 = [Z_3^2], \quad X_3 = [B^2 + C \cdot [A^2]] + D, \quad Y_3 = [(X_3 + [X_2 \cdot Z_3]) \cdot D + (X_2 + Y_2) \cdot T_3]. \]

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
The scalar multiplication on the curve $E(F_{2^m})$ consists in the computation of $r \cdot P$ for a given point $P \in E(F_{2^m})$ and an $\ell$-bit integer $r$ where $\ell$ 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 operation.
algorithm. The scalar multiplication then requires a table precomputation $T[i] = i \cdot P$ for the odd integers $0 < i < 2^{w-1}$. This approach is detailed in Algorithm 7 in the appendix. In our implementations we used the Kim-Kim (cf. Subsection 4.1.1) and the Lopez-Dahab (Table 6) 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:

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

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 given Algorithm 8 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], 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. Details are given in Algorithm 9 in the appendix.

- **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 $S_q$ 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_ intrinsic function with left input value $S_q$ 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

$$\sqrt{A} = \left( \sum_{i=0}^{[m/2]} a_{2i} X^i \right) + \sqrt{x} \left( \sum_{i=0}^{[m/2]} a_{2i+1} X^i \right).$$

11
Following [1], we separate odd and even coefficients of $A$ using the \texttt{mm_shuffle_epi8} intrinsic function 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:

$$HT(A) = \sum_{i=0}^{(m-1)/2} A^{2^i}.$$

Our implementation is again inspired from [16] and [7] and uses the \texttt{mm_shuffle_epi8} function 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].

**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, Table 6 and 7).

- **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 1 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 $c_{255}, \ldots, c_{233}$.

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 8. 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.

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 $ABplusCD$.

Based on the results reported in Table 8, 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 9 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 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
Table 8: Timings in terms of $10^3$ clock-cycles of scalar multiplication on an Intel Core 2 (2.50GHz)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Formulas</th>
<th>$m = 233$</th>
<th>$m = 409$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(#CC)/10³</td>
<td>(#CC)/10³</td>
</tr>
<tr>
<td>Double-and-add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>KK</td>
<td>592</td>
<td>2125</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>613</td>
<td>2192</td>
</tr>
<tr>
<td>LR1</td>
<td>KK</td>
<td>1249</td>
<td>2207</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>1179</td>
<td>2832</td>
</tr>
<tr>
<td>AB, AC</td>
<td>KK</td>
<td>558</td>
<td>6217</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>928</td>
<td>2917</td>
</tr>
<tr>
<td>ABplusCD</td>
<td>KK</td>
<td>542</td>
<td>2187</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>553</td>
<td>2206</td>
</tr>
<tr>
<td>Halve-and-add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>KK</td>
<td>387</td>
<td>1504</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>403</td>
<td>1575</td>
</tr>
<tr>
<td>LR1</td>
<td>KK</td>
<td>651</td>
<td>1706</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>855</td>
<td>1837</td>
</tr>
<tr>
<td>AB, AC</td>
<td>KK</td>
<td>858</td>
<td>2277</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>887</td>
<td>2359</td>
</tr>
<tr>
<td>ABplusCD</td>
<td>KK</td>
<td>375</td>
<td>1504</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>386</td>
<td>1640</td>
</tr>
<tr>
<td>Parallel(*) (Double-and-add + Halve-and-add)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>KK</td>
<td>280</td>
<td>965</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>295</td>
<td>999</td>
</tr>
<tr>
<td>LR1</td>
<td>KK</td>
<td>335</td>
<td>1042</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>315</td>
<td>1104</td>
</tr>
<tr>
<td>AB, AC</td>
<td>KK</td>
<td>270</td>
<td>2311</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>289</td>
<td>1382</td>
</tr>
<tr>
<td>ABplusCD</td>
<td>KK</td>
<td>273</td>
<td>977</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>277</td>
<td>1014</td>
</tr>
<tr>
<td>Montgomery</td>
<td>none</td>
<td>-</td>
<td>593</td>
</tr>
<tr>
<td></td>
<td>LR1</td>
<td>-</td>
<td>2190</td>
</tr>
<tr>
<td></td>
<td>ABplusCD</td>
<td>-</td>
<td>2482</td>
</tr>
</tbody>
</table>

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

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 $m = 409$.

In Table 10, we review the results obtained by Aranha et al. over an Intel Core i5 with a GCC compiler. We remark that, except for parallel implementation when $m = 409$, our results are competitive with the timings of Table 10. This means that our implementations reach the level of performance of [16] 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
Table 9: Timings in terms of $10^3$ clock-cycles of scalar multiplication on an Intel Core i5 (2.5 GHz)

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Curve Formulas</th>
<th>$m = 233$</th>
<th>$m = 409$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#CC/10³</td>
<td>#CC/10³</td>
</tr>
<tr>
<td>Double-and-add</td>
<td>none</td>
<td>KK</td>
<td>246</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>LR1 and LR2(**)</td>
<td>KK</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>228</td>
</tr>
<tr>
<td></td>
<td>$AB, AC$ and</td>
<td>KK</td>
<td>219</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>226</td>
</tr>
<tr>
<td></td>
<td>$ABplusCD$ and</td>
<td>KK</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>222</td>
</tr>
<tr>
<td>Halve-and-add</td>
<td>none</td>
<td>KK</td>
<td>165</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>169</td>
</tr>
<tr>
<td></td>
<td>LR1 and LR2(**)</td>
<td>KK</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>$AB, AC$ and</td>
<td>KK</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>$ABplusCD$ and</td>
<td>KK</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>154</td>
</tr>
<tr>
<td>Parallel(*)</td>
<td>none</td>
<td>KK</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>LR1 and LR2(**)</td>
<td>KK</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>$AB, AC$ and</td>
<td>KK</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>$ABplusCD$ and</td>
<td>KK</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td>118</td>
</tr>
<tr>
<td>Montgomery</td>
<td>none</td>
<td>-</td>
<td>244</td>
</tr>
<tr>
<td></td>
<td>LR1 and LR2(**)</td>
<td>-</td>
<td>229</td>
</tr>
<tr>
<td></td>
<td>$ABplusCD$ and</td>
<td>-</td>
<td>220</td>
</tr>
</tbody>
</table>

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

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 $F_{233}$. On an Intel Core i5, the optimization provides the best results for scalar multiplication over the two considered fields $F_{233}$ and $F_{409}$. 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 PAVOIS ANR 12 BS02 002 02.
Table 10: Timing results of best know results [16] obtained with GCC on Intel Core i5

<table>
<thead>
<tr>
<th>Method</th>
<th>B233</th>
<th>B409</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-and-add, $w = 4$</td>
<td>231</td>
<td>941</td>
</tr>
<tr>
<td>Halve-and-add, $w = 4$</td>
<td>188</td>
<td>706</td>
</tr>
<tr>
<td>Parallel</td>
<td>122</td>
<td>444</td>
</tr>
</tbody>
</table>

References


Algorithm 6 CombMul_G_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 $(k = 2; k < 16; k += 2)\{$
  $T[k][0] = (T[k] >> 1)[0] << 1;$
  $T[k + 1][0] = T[k][0] \land A[0];$
  for $(i = 1; i < N; i + +)\{$
    $T[k][i] = (T[k] >> 1)[i] << 1$
    $\land (T[k] >> 1)[i - 1] >> 63;$
    $T[k + 1][i] = T[k][i] \land A[i];$
  \}

for $(k = 15; k >= n - 16(N - 1); k -= )\{$
  for $(j = 0; j < N - 1; j + +)\{$
    $u = (B[j] >> (4 * k)) \& 0xf$
    $R[i + j] = R[i + j] \land T[u][i];$
  \}
  carry = 0
  for $(i = 0; i < 2 * N; i + +)\{$
    temp = $R[i];$
    $R[i] = (R[i] << 4) \land carry;$
    carry = temp >> 60;
  \}

for $(k = n - 16(N - 1) - 1; k > 0; k -= )\{$
  for $(j = 0; j < N - 1; j + +)\{$
    $u = (B[j] >> (4 * k)) \& 0xf$
    $R[i + j] = R[i + j] \land T[u][i];$
  \}
  carry = 0
  for $(i = 0; i < 2 * N; i + +)\{$
    temp = $R[i];$
    $R[i] = (R[i] << 4) \land carry;$
    carry = temp >> 60;
  \}

for $(j = 0; j < N; j + +)\{$
  $u = B[j] \& 0xf$
  for $(i = 0; i < N - 1; i + +)\{$
    $R[i + j] = (R[i + j] << 4) \land T[u];$
  \}
\}
Algorithm 7 Double-and-add with NAF<sub>W</sub>

**Require:** Window width \( w \), \( P \in E(\mathbb{F}_{2^w}) \) and an integer \( k \).

**Ensure:** \( Q = k \cdot P \).

1. Compute \( NAF_{W}(k) = \sum_{i=0}^{l-1} k_i \cdot 2^i \).
2. Compute \( P_i = i \cdot P \) for \( i \in \{1, 3, 5, ..., 2^w - 1\} \).
3. \( Q \leftarrow O \).
4. for \( i \) from \( l - 1 \) downto 0 do
5. \( Q \leftarrow 2 \cdot Q \).
6. if \( k_i > 0 \) then
7. \( Q \leftarrow Q + P_{k_i} \).
8. end if
9. if \( k_i < 0 \) then
10. \( Q \leftarrow Q - P_{-k_i} \).
11. end if
12. end for
13. return \( Q \)

Algorithm 8 Halve-and-add with NAF<sub>W</sub>

**Require:** Window width \( w \), \( P \in E(\mathbb{F}_{2^w}) \) and an integer \( k \).

**Ensure:** \( Q = k \cdot P \).

1. \( k' = k \cdot 2^w \mod \#E(\mathbb{F}_{2^w}) \).
2. Compute \( NAF_{W}(k') = \sum_{i=0}^{l-1} k'_i \cdot 2^i \).
3. \( R \leftarrow P \).
4. for \( i \) from \( l - 1 \) downto 0 do
5. if \( k_i' > 0 \) then
6. \( Q_{k_i'} \leftarrow Q_{k_i'} + R \).
7. end if
8. if \( k_i' < 0 \) then
9. \( Q_{-k_i'} \leftarrow Q_{-k_i'} - R \).
10. end if
11. \( R \leftarrow R/2 \).
12. end for
13. \( Q \leftarrow \sum_{i \in \{1, 3, 5, ..., 2^w - 1\}} i \cdot Q_i \).
14. return \( Q \)

Algorithm 9 Parallel (Double-and-add, Halve-and-add) with NAF<sub>W</sub>

**Require:** Window width \( w \), \( P \in E(\mathbb{F}_{2^w}) \) and an integer \( k \).

**Ensure:** \( Q = k \cdot P \).

\( k' = k \cdot 2^t \mod \#E(\mathbb{F}_{2^w}) \)

Compute \( NAF_{W}(k') = \sum_{i=0}^{l'} k'_i \cdot 2^i \)

// ———— Begin parallel execution ————

Compute \( P_i = i \cdot P \) for \( i \in \{1, 3, 5, ..., 2^w - 1\} \)

\( Q \leftarrow O \)

for \( i \) from \( t \) downto \( 0 \)

\( Q \leftarrow 2 \cdot Q \)

if \( k_i' > 0 \)

\( Q \leftarrow Q + P_{k_i'} \)

end if

if \( k_i' < 0 \)

\( Q \leftarrow Q - P_{-k_i'} \)

end if

end for

// ———— End parallel execution ————

\( Q \leftarrow Q + \sum_{i \in \{1, 3, 5, ..., 2^w - 1\}} i \cdot Q_i \)

**return** \( Q \leftarrow Q + \sum_{i \in \{1, 3, 5, ..., 2^w - 1\}} i \cdot Q_i \)