Efficient Multiplication in $GF(p^k)$ for Elliptic Curve Cryptography

J.-C. Bajard, L. Imbert, C. Nègre and T. Plantard
Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier
LIRMM, 161 rue Ada, 34392 Montpellier cedex 5 – France
{bajard, imbert, negre, plantard}@lirmm.fr

Abstract

We present a new multiplication algorithm for the implementation of elliptic curve cryptography (ECC) over the finite extension fields $GF(p^k)$ where p is a prime number greater than $2k$. In the context of ECC we can assume that p is a 7-to-10-bit number, and easily find values for k which satisfy: $p > 2k$, and for security reasons $\log_2(p) \times k \simeq 160$. All the computations are performed within an alternate polynomial representation of the field elements which is directly obtained from the inputs. No conversion step is needed. We describe our algorithm in terms of matrix operations and point out some properties of the matrices that can be used to improve the design. The proposed algorithm is highly parallelizable and seems well adapted to hardware implementation of elliptic curve cryptosystems.

1. Introduction

Cryptographic applications such as elliptic or hyperelliptic curves cryptosystems (ECC, HECC) [11, 12, 13] require arithmetic operations to be performed in finite fields. This is the case, for example, for the Diffie-Hellman key exchange algorithm [6] which bases its security on the discrete logarithm problem. Efficient arithmetic in these fields is then a major issue for lots of modern cryptographic applications [14]. Many studies have been proposed for the finite field $GF(p)$, where p is a prime number [23] or the Galois field $GF(2^k)$ [4, 7, 16]. In [1], D. Bailey and C. Paar use optimal extension fields $GF(p^k)$ and they propose an efficient arithmetic solution in those fields when p is a Mersenne or pseudo-Mersenne prime [2]. Although it could result in a wider choice of cryptosystems, arithmetic over the more general finite extension fields $GF(p^k)$, with $p > 2$, has not been extensively investigated yet. Moreover it has been proved that elliptic curves defined over $GF(p^k)$ – where the curves verify the usual conditions of security – provide at least the same level of security as the curves usually defined over $GF(2^k)$ or $GF(p)$. For ECC, a good level of security can be achieved with p and k prime and about 160-bit key-length. Table 1 gives some good candidates for p and k and the corresponding key-size.

p	k	key-size	form of p
67	29	175	$64+3$
67	31	188	$= 1000011$
73	23	142	$64+8+1$
73	29	179	$= 1001001$
127	19	132	$128-1$
127	23	160	$= M_7 = 2^7-1$
127	29	202	$= 1111111$
257	17	136	$256+1$
257	19	152	$= F_4 = 2^{2^3}+1$
257	23	184	$= 100000001$

Table 1. Good candidates for primes p and k and the corresponding key-size in bits.

In this paper, we introduce a Montgomery like modular multiplication algorithm in $GF(p^k)$ for $p > 2k$ (this condition comes from technical reasons that we shall explain further). Given the polynomials $A(X)$ and $B(X)$ of degree less than k, and $G(X)$ of degree k (we will give more details on $G(X)$ in section 1.2), our algorithm computes

$$A(X)\,B(X)\,G(X)^{-1} \mod N(X),$$

where $N(X)$ is a monic irreducible polynomial of degree k ; and both the operands and the result are given in an alternate representation introduced in the next section.

In the classical polynomial representation, we can consider the elements of $GF(p^k)$ as polynomials of degree less than k in $GF(p)[X]$ and we represent the field with respect to an irreducible polynomial $N(X)$ of degree k over $GF(p)$. Any element A of $GF(p^k)$ is then represented using a polynomial $A(X)$ of degree $k-1$ or less with coefficients in $GF(p)$, i.e., $A(X) = a_0 + a_1X + \cdots + a_{k-1}X^{k-1}$, where $a_i \in \{0, \ldots, p-1\}$. Here, we consider an alternate solution which consists of representing the polynomials with their values at k distinct points instead of their k coefficients. As a result, if we choose k points $(e_1, e_2, \ldots, e_k)$, we represent the polynomial A with the sequence $(A(e_1), A(e_2), \ldots, A(e_k))$. Within this representation, addition, subtraction and multiplication are performed over completely independent channels which has great advantage from a chip design viewpoint.

1.1. Montgomery Multiplication in $GF(p^k)$

Montgomery's technique for modular multiplication of large integers [15] has recently been adapted to modular multiplication in $GF(2^k)$ by Koç and Acar [4]. The proposed solution is a direct translation of the original Montgomery algorithm in the field $GF(2^k)$, with X^k playing the role of the Montgomery factor ; i.e. it computes $A(X)\,B(X)\,X^{-k} \bmod N(X)$, where N is a k-order irreducible polynomial with coefficients in $GF(2)$.

In turn this method easily extends to $GF(p^k)$, with $p > 2$. As in [4], we represent the field $GF(p^k)$ with respect to a monic irreducible polynomial $N(X)$ and we consider the field elements as polynomials of degree less than k in $GF(p)[X]$; i.e. we consider the elements of $GF(p)[X]/N(X)$. Thus, if we take A and B in $GF(p^k)$, we successively compute

$$\begin{array}{rcl} Q(X) & = & -A(X)\,B(X)\,N(X)^{-1} \bmod X^k \\ R(X) & = & [A(X)\,B(X) + Q(X)\,N(X)]\;X^{-k} \end{array}$$

to get the result $A(X)\,B(X)\,X^{-k} \bmod N(X)$. In terms of elementary operations over $GF(p)$, the complexity of this method is $k^2 + (k-1)^2$ multiplications (modulo p) and $(k-1)^2 + (k-2)^2 + k$ additions (modulo p).

1.2. Alternate polynomial representation

The general idea of our approach is the change of representation. When dealing with polynomials the idea which first comes in mind is to use a coefficient representation. However, the valued representation – where a polynomial is represented by its values at sufficiently many points – can be of use. Thanks to Lagrange's theorem we can actually represent any polynomial of degree less than k with its values at various distinct points $\{e_1, e_2, \ldots, e_k\}$. A very good discussion on polynomial evaluation and interpolation can be found in [21].

In the following of the paper, we represent a polynomial of degree at most $k-1$, say A, by the sequence $(A(e_1), A(e_2), \ldots, A(e_k))$. In the following we consider the notation $a_i = A(e_i)$. At this point, it is very important to understand that the a_is do not represent the coefficients of A, and that there is nothing to do to obtain such a representation. We directly consider the polynomial in this form. As an example, the input 100111010101 which would usually represent the polynomial $9X^2 + 13X + 5$ in the coefficient representation, is considered here as the sequence $(9, 13, 5)$. This sequence corresponds to the unique polynomial P of degree 2 which has values $P(e_1) = 9$, $P(e_2) = 13$ and $P(e_3) = 5$. We can easily compute its coefficients by means of interpolation but as we shall see further, there is no need to do so. We will use this representation during all the computational steps.

2. New algorithm

As mentioned previously, Koç and Acar used the polynomial X^k in their adaptation of Montgomery multiplication to the field $GF(2^k)$, and we have briefly shown in section 1.1 that their solution easily extends to $GF(p^k)$. In our new approach we rather consider the k-order polynomial

$$G(X) = (X - e_1)(X - e_2)\ldots(X - e_k), \qquad (1)$$

where $e_i \in \{0, 1, \ldots, p-1\}$. A first remark is that this clearly implies $p > k$. As we shall see further, $2k$ distinct points are actually needed. Thus given the three polynomials $A = (a_1, a_2, \ldots, a_k)$, $B = (b_1, b_2, \ldots, b_k)$ and $N = (n_1, n_2, \ldots, n_k)$ in $GF(p^k)$; and under the condition $p > 2k$, our algorithm computes the product $A(X)\,B(X)\,G^{-1}(X) \mod N(X)$ in two stages.

Stage 1: We define the polynomial Q of degree less than k such that:

$$Q(X) = \left[-A(X)\,B(X)\,N^{-1}(X)\right] \bmod G(X),$$

in other words, we compute in parallel and in $GF(p)$

$$q_i = \left[-a_i\,b_i\,N^{-1}(e_i)\right], \quad \forall i = 1 \ldots k.$$

Stage 2: Since $[A(X)\,B(X) + Q(X)\,N(X)]$ is a multiple of $G(X)$, we compute $R(X)$ of degree less than k such that

$$R(X) = \Big[A(X)\,B(X) + Q(X)\,N(X)\Big]\,G^{-1}(X)$$

In this algorithm it is important to note that it is not possible to evaluate $R(X)$ directly as mentioned in step 2. Since $[A(X)\,B(X) + Q(X)\,N(X)]$ is a multiple of $G(X)$ its representation at the points $\{e_1, e_2, \ldots, e_k\}$ is merely composed of 0. The same clearly applies for $G(X) = \prod_{i=1}^{k}(X - e_i)$. As a direct consequence the division by $G(X)$, which actually reduces to the multiplication by $G^{-1}(X)$, has neither effect nor sense. We address this problem by using k extra values $\{e'_1{}_2,e'\ldots, e'_k\}$ where $e'_i \neq e_j$ for all i, j, and by computing $[A(X)\,B(X) + Q(X)\,N(X)]$ for those k extra values. In algorithm 1, the operations in step 3 are then performed for $X \in \{e'_1{}_2,e'\ldots, e'_k\}$.

Steps 1 and 3 are fully parallel operations in $GF(p)$. The complexity of algorithm 1 thus mainly depends on the two polynomial interpolations (steps 2, 4).

Algorithm 1 New Multiplication in $GF(p^k)$

```
Step 1: For $X \in \{e_1, \ldots, e_k\}$, compute in parallel
    $$Q(X) = -A(X)\,B(X)\,N^{-1}(X)$$
Step 2: Extend $Q$ in $\{e'_1, \ldots, e'_k\}$ using Lagrange interpolation
Step 3: For $X \in \{e'_1, \ldots, e'_k\}$, compute in parallel
    $$R(X) = \Big[A(X)B(X) + Q(X)N(X)\Big]G^{-1}(X)$$
Step 4: Extend $R$ back in $\{e_1, \ldots, e_k\}$ using Lagrange interpolation.
```

2.1. Implementation

In step 1 we compute in $GF(p)$ and in parallel for all i in $\{1, \ldots, k\}$

$$q_i = (-a_i \times b_i \times \tilde{n}_i) \bmod p, \qquad (2)$$

where the $\tilde{n}_i$s are precomputed constants ($\tilde{n}_i = N^{-1}(e_i)$). Then in step 2, the extension is performed via Lagrange interpolation:

$$Q(X) = \sum_{i=1}^{k} q_i \left(\prod_{j=1, j\neq i}^{k} \frac{X - e_j}{e_i - e_j} \right). \qquad (3)$$

If we denote

$$\omega_{t,i} = \prod_{j=1, j\neq i}^{k} \frac{e'_t - e_j}{e_i - e_j}, \qquad (4)$$

the extension of $Q(X)$ in $\{e'_{1\,2}e'\ldots, e'_k\}$ becomes

$$\begin{pmatrix} q'_1 \\ q'_2 \\ \vdots \\ q'_{k-1} \\ q'_k \end{pmatrix} = \begin{pmatrix} \omega_{1,1} & \cdots & \omega_{1,k-1} & \omega_{1,k} \\ \omega_{2,1} & \cdots & \omega_{2,k-1} & \omega_{2,k} \\ \vdots & & & \\ \omega_{k-1,1} & \cdots & \omega_{k-1,k-1} & \omega_{k-1,k} \\ \omega_{k,1} & \cdots & \omega_{k,k-1} & \omega_{k,k} \end{pmatrix} \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_{k-1} \\ q_k \end{pmatrix}. \qquad (5)$$

Operations in step 3 are performed in parallel for $i \in \{1, \ldots, k\}$. We compute

$$r'_i = (a'_i \times b'_i + q'_i \times n'_i)\,\zeta_i \bmod p, \qquad (6)$$

where the ζ_i s are also precomputed values.

$$\zeta_i = G(e'_i)^{-1} \bmod p = \left[\prod_{j=1}^{k} (e'_i - e_j) \right]^{-1} \bmod p.$$

It is easy to remark that $G(e'_i) \neq 0, \forall i \in \{1, \ldots, k\}$. Thus the modular inverse, $G(e'_i)^{-1} \bmod p$, always exists.

At the end of step 3, the polynomial R of degree less than k is defined by its values at $\{e'_{1\,2}e'\ldots, e'_k\}$, namely $(r'_1, r'_2, \ldots, r'_k)$. If we want to reuse the obtained result as the input of other multiplications (which is frequently the case in exponentiation algorithms), we must also know the values of R at $\{e_1, e_2, \ldots, e_k\}$. This is done in step 4 again by mean of Lagrange interpolation. As in step 2, we define

$$\omega'_{t,i} = \prod_{j=1, j\neq i}^{k} \frac{e_t - e'_j}{e'_i - e'_j}, \qquad (7)$$

and we compute

$$\begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_{k-1} \\ r_k \end{pmatrix} = \begin{pmatrix} \omega'_{1,1} & \cdots & \omega'_{1,k-1} & \omega'_{1,k} \\ \omega'_{2,1} & \cdots & \omega'_{2,k-1} & \omega'_{2,k} \\ \vdots & & & \\ \omega'_{k-1,1} & \cdots & \omega'_{k-1,k-1} & \omega'_{k-1,k} \\ \omega'_{k,1} & \cdots & \omega'_{k,k-1} & \omega'_{k,k} \end{pmatrix} \begin{pmatrix} r'_1 \\ r'_2 \\ \vdots \\ r'_{k-1} \\ r'_k \end{pmatrix}. \qquad (8)$$

Another implementation option would be to insert some of the multiplications by constants into the matrix operations of steps 2 and 4. We can introduce the $\tilde{n}_i$s of (2) and the n'_is of (6) in the matrix of equation (5) to gain one product in each step 1 and 3. We do not give much details about this solution because we will see further that the original matrices have some very attractive properties for the hardware implementation.

2.2. Example

In this example, we consider the finite field $GF(17^5)$ defined according to the monic irreducible polynomial $N(X) = X^5 + 4X + 1$. ($p = 17$ and $k = 5$ satisfy $p > 2k$.) The two sets of points used for Lagrange representation are $\mathsf{E} = \{2, 4, 6, 8, 10\}$ and $\mathsf{E}' = \{3, 5, 7, 9, 11\}$. For all e_i in E and e'_i in E', we have $N^{-1}(e_i) = (5,\ 13,\ 8,\ 15,\ 4)$ (for use in step 1) and $N(e'_i) = (1,\ 1,\ 6,\ 11,\ 4)$ (for use in step 3). Also used in step 3 is the vector $\zeta = (6,\ 3,\ 14,\ 11,\ 12)$. The two interpolation matrices needed in steps 2 and 4 are:

$$\omega = \begin{pmatrix} 2 & 8 & 13 & 5 & 7 \\ 7 & 1 & 10 & 11 & 6 \\ 6 & 11 & 10 & 1 & 7 \\ 7 & 5 & 13 & 8 & 2 \\ 2 & 14 & 8 & 10 & 1 \end{pmatrix}$$

and

$$\omega' = \begin{pmatrix} 1 & 10 & 8 & 14 & 2 \\ 2 & 8 & 13 & 5 & 7 \\ 7 & 1 & 10 & 11 & 6 \\ 6 & 11 & 10 & 1 & 7 \\ 7 & 5 & 13 & 8 & 2 \end{pmatrix}.$$

In Lemma 1 we will observe some symmetry between the elements of these two matrices.

Given $A(X)$ and $B(X)$ in $GF(17^5)$, known by their values at points of E and E', we compute $R(X) = A(X)B(X)G^{-1}(X) \bmod N(X)$ in the same representation. We have:

$$\begin{aligned} A(\mathsf{E}) &= (3,\ 9,\ 0,\ 9,\ 4) \\ A(\mathsf{E}') &= (15,\ 0,\ 1,\ 10,\ 5) \\ B(\mathsf{E}) &= (5,\ 3,\ 12,\ 12,\ 0) \\ B(\mathsf{E}') &= (12,\ 1,\ 8,\ 13,\ 13) \end{aligned}$$

In step 1 of the algorithm we compute

$$Q(\mathsf{E}) = (10,\ 6,\ 0,\ 12,\ 0)$$

and we extend it in step 2 (eq. (5)) from E to E'

$$Q(\mathsf{E}') = (9,\ 4,\ 2,\ 9,\ 3)$$

Now in step 3 (eq. (6)), we evaluate in parallel for each value of E'

$$R(\mathsf{E}') = (12,\ 12,\ 8,\ 3,\ 6)$$

and we interpolate it back (eq. (8)) to obtain the final result in E

$$R(\mathsf{E}) = (12,\ 9,\ 7,\ 6,\ 12).$$

We can easily check that this actually is the correct result. If we consider the classical coefficient representation of A and B, we have $A(X) = 2\,X^4 + X + 3,\ B(X) = X^3 + 5\,X + 4$ and $R(X) = 4\,X^4 + 5\,X^3 + 14\,X^2 + 5$, which evaluated at points of E gives $R(\mathsf{E})$.

3. Arithmetic over $GF(p)$

From an hardware point of view, this method is of interest if and only if we can take advantage of an efficient arithmetic over $GF(p)$. In this section we give the idea of some algorithms for the addition and the multiplication modulo a prime p. Different solutions have been proposed but most of them only focus on large primes which are useful if one wants to implement elliptic curve cryptography over $GF(p)$ [23]. Here we only need arithmetic operations modulo small primes, say 8 to 12 bits.

3.1. Addition

When we aim at computing the modular sum $a + b \bmod m$, a classical approach consists in evaluating in parallel the quantities $a + b$ and $a + b - m$. The correct result is selected according to the sign of $a + b - m$. For a single operation, this solution gives a result less than m. However, when several additions have to be computed, we do not need to reduce the sum modulo m after each addition. If $2^{m-1} \leqslant m < 2^m$, another solution is to keep the intermediate results less than 2^m by only performing a reduction modulo m when the partial sum becomes greater than 2^m. In other words, we perform the sum $a + b$ and we subtract m only if a carry has occurred. In [19], a redundant representation is used so that the modular addition is performed without carry propagation. The redundant addition is then used within a radix-2 and radix-4 modular multiplication algorithms.

3.2. Multiplication

Multiplication modulo special numbers have been extensively studied. For instance a multiplication modulo $2^n - 1$ is presented in [17] and modulus of the form 2^n+1, are used in [22] in the context of DSP applications. Other works exists for Fermat numbers $F_n = 2^{2^n} + 1$ and Mersenne primes $M_p = 2^p - 1$, with p prime.

In the general case, the product $a \times b \bmod p$ can be implemented by means of index calculus with two lookup tables and one addition. We simply use the fact that any element of the group $GF(p)$ corresponds to a power of a generator g of the group. We retrieve in a first table the values α and β such that $a = g^\alpha$ and $b = g^\beta$, we evaluate $\alpha + \beta \bmod p - 1$ and we read in the second table the result $r = a\,b \bmod p = g^{|\alpha+\beta|_{p-1}}$ (see figure 1). This solution has been proposed in [9] for the special case of the 4th Fermat prime $p = 257$. The advantage here is that addition modulo $p - 1$ reduces to a classical 8-bit addition.

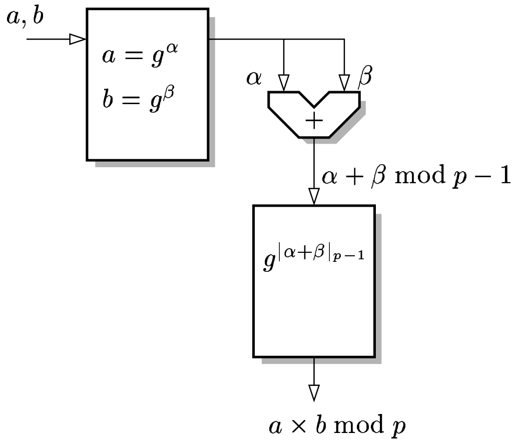

Figure 1. Index calculus modular multiplication.

An interesting suggestion for multiplication by means of look-up tables can be found in [20] under the term quarter-squarer multiplier. It is based of the following equation:

$$a\,b \bmod n = \left(\left(\frac{a+b}{2}\right)^2 - \left(\frac{a-b}{2}\right)^2\right) \bmod n, \qquad (9)$$

where both squares are given by a look-up table of 2^{n+1} input bits. This is illustrated in figure 2. Optimizations of this general idea are possible. One can perform the division by two (shifts) before the table look-up. This divides the size of the table by a factor two, but when $a+b$ is odd, the correcting term b must be added (modulo p) at the end. The size can be further reduced using the fact that $x^2 \equiv (-x)^2 \pmod{p}$. In this case, the address resolution problem must be solved.

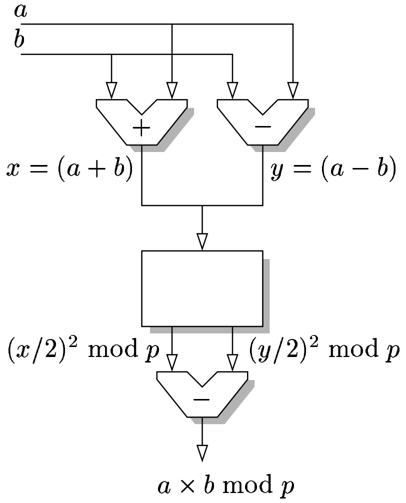

Figure 2. Look-up based modular multiplication for small operands.

One can also consider a double-and-add method, sometimes called the Russian peasant method for multiplication [10], associated with a Booth recoding of one of the operands. A table is used to store the double (modulo p) of each value less than p. If we want to multiply two m-bit numbers, this method requires at most $m/2$ additions. For example the evaluation of $121 \times 17 \bmod 29$ only requires 2 additions and 7 doublings which are just table lookups. $|121 \times 17|_{29} = 2(2(2(2(2(2(2 \times 17 \bmod 29) \bmod 29) \bmod 29) - 17 \bmod 29) \bmod 29) \bmod 29) + 17 \bmod 29$

Modular multiplication by a constant is a lot easier. For small operands, one can simply implement the modular multiplication with some combinatorial logic implementing the function.

4. Complexity

In table 2 we count the number of additions (A), multiplications by a constant (CM) and real multiplications (M) over $GF(p)$ of algorithm 1.

The time required for a sequential implantation corresponds to the number of operations given in table 2. Since the product in $GF(p^k)$ can be totally parallelized into k streams, the time required is exactly $1/k$ times that for the sequential version. If we define T_A the time required for one addition, T_{CM} for one constant multiplication and T_M for one real multiplication respectively, we can precisely evaluate the time complexity of our algorithm on a parallel architecture. Table 3 summarizes the four steps of the algorithm.

	A	CM	M
step 1	-	k	k
step 2	$k(k-1)$	k^2	-
step 3	k	k	$2k$
step 4	$k(k-1)$	k^2	-
total	$2k^2 - k$	$2k^2 + 2k$	$3k$

Table 2. Number of additions (A), constant multiplications (CM) and real multiplications (M) over $GF(p)$ for a sequential implementation of the algorithm.

step 1	$T_M + T_{CM}$
step 2	$T_{CM} + \sum_{j=1}^{k-1} \max(T_{CM}, T_A)$
step 3	$\max(T_M, T_{CM}) + T_A + T_{CM}$
step 4	$T(MC) + \sum_{j=1}^{k-1} \max(T_{CM}, T_A)$

Table 3. Time complexity estimation on a pipelined architecture.

5. Discussion

5.1. Simplified architecture

The major advantage of this method is that the matrices in (5) and (8) do not depend on the inputs. Thus all the operations reduce to multiplications by constants which significantly simplify the hardware implementation. Moreover, in the example presented in section 2.2 we have detected symmetries between the elements of the two matrices that can also contribute to a simplified architecture. We have the following Lemma.

Lemma 1 *As in the previous example, let us denote* $e_i = 2i$ *and* $e'_j = 2j+1$. *According to equations (4) and (7) we have*

$$\omega_{i,j} = \prod_{m=1, m\neq j}^{k} \frac{2i+1-2m}{2j-2m} \tag{10}$$

and

$$\omega'_{i,j} = \prod_{m=1, m\neq j}^{k} \frac{(2i-(2m+1))}{(2j+1-(2m+1))}. \tag{11}$$

Then for every $i, j \in \{1, \dots, k\}$ we have

$$\omega_{i,j} = \omega'_{k+1-i,k+1-j}. \tag{12}$$

In other words equation (8) can be implemented with the same matrix than eq. (5), by simply reversing the order of the elements of the vectors r and r':

$$\begin{pmatrix} r_k \\ r_{k-1} \\ \vdots \\ r_2 \\ r_1 \end{pmatrix} = \begin{pmatrix} \omega_{1,1} & \cdots & \omega_{1,k-1} & \omega_{1,k} \\ \omega_{2,1} & \cdots & \omega_{2,k-1} & \omega_{2,k} \\ \vdots & & & \\ \omega_{k-1,1} & \cdots & \omega_{k-1,k-1} & \omega_{k-1,k} \\ \omega_{k,1} & \cdots & \omega_{k,k-1} & \omega_{k,k} \end{pmatrix} \begin{pmatrix} r'_k \\ r'_{k-1} \\ \vdots \\ r'_2 \\ r'_1 \end{pmatrix}. \tag{13}$$

Proof: *We are going to rearrange each part of the equality to make the identity appear. Let us first focus on the right-hand part of the identity.*

$$\begin{aligned} &\omega'_{k+1-i,k+1-j} = \\ &\prod_{m \neq k+1-j} \frac{2(k+1-i)-(2m+1)}{2(k+1-j)+1-(2m+1)} \\ &= \prod_{m \neq k+1-j} \frac{-2i+2(k+1-m)-1}{-2j+1+2(k+1-m)-1}. \end{aligned}$$

So far we have just changed the position of $k+1$ in each term of the product. Next just by multiplying each fraction by -1, and extracting all the 2s in the denominators, we get:

$$\begin{aligned} &\omega'_{k+1-i,k+1-j} = \\ &2^{1-k} \prod_{m \neq k+1-j} \frac{2i-2(k+1-m)+1}{j-(k+1-m)} \\ &= 2^{1-k} \prod_{m \neq j} \frac{2i+1-2m}{j-m}. \end{aligned}$$

Here we have reordered the indices $m \leftarrow k+1-m$. We now do the same with the left-hand expression.

$$\omega_{i,j} = \prod_{m \neq j} \frac{2i+1-2m}{2j-2m}.$$

We extract the 2s in the denominators:

$$\omega_{i,j} = 2^{1-k} \prod_{m \neq j} \frac{2i+1-2m}{j-m},$$

and we conclude that the new expressions for $\omega_{i,j}$ and $\omega'_{k+1-i,k+1-j}$ are the same. □

This lemma points out symmetry properties of the matrices that mainly depend on the choice made in the example for the points of e and e'. They can be taken into account to improve the hardware architecture. Other choices of points could be more interesting and could result in very attractive chip design solutions. This is currently work in progress in our team.

5.2. Cryptographic context

In ECC, the main operation is the addition of two points of an elliptic curve defined over a finite field. Hardware implementation of elliptic curves cryptosystems thus requires efficient operators for additions, multiplications and divisions. Since division is usually a complex operation, we use projective (or homogeneous) coordinates to bypass this difficulty (only one division is needed at the very end of the algorithm).

Thus the only operations are addition and multiplication in $GF(p)$. Moreover it is worth noticing that we do not need to reduce modulo p after each addition. We only subtract p from the result of the last addition if it is greater than $2^{\lceil log_2(p) \rceil}$ (we recall that p is odd). In other words we just have to check one bit after each addition. The exact value is only needed for the final result.

In ECC protocols, additions chains of points of an elliptic curve are needed. In homogeneous coordinates, those operations consist in additions and multiplications over $GF(p^k)$. Only one division is needed at the end and it can be performed in the Lagrange representation using the Fermat-Euler theorem which states that for all non zero value x in $GF(p^k)$, then $x^{p^k-1} = 1$. Hence we can compute the inverse of x by computing x^{p^k-2} in $GF(p^k)$.

It is also advantageous to use a polynomial equivalent to the Montgomery notation during the computations. We consider polynomials in the form

$$A'(X) = A(X)\,G(X) \bmod N(X)$$

instead of $A(X)$. It is clear that adding two polynomials given in this notation gives the result in the same notation, and for the product, since

$$\text{Mont}(A, B, N) = A(X)\,B(X)\,G^{-1}(X) \bmod N(X),$$

we have

$$\begin{aligned} &\text{Mont}(A', B', N) = \\ &\qquad A'(X)\,B'(X)\,G^{-1}(X) \bmod N(X) \\ &\qquad = A(X)\,B(X)\,G(X) \bmod N(X). \end{aligned}$$

6. Conclusion

Works from Bailey and Paar [1, 2], Smart [18] and Crandall [5] have shown that it is possible to obtain more efficient software implementation over $GF(p^k)$ than over $GF(2^k)$ or $GF(p)$ when p is carefully chosen (Mersenne, pseudo-Mersenne, generalized Mersenne primes, etc). In this article we have presented a new modular multiplication algorithm over the finite extension field $GF(p^k)$, for $p > 2k$, which is highly parallelizable and well adapted to hardware implementation. Our algorithm is particularly interesting for ECC since it seems that there exists fewer non-singular curves over $GF(p^k)$ than over $GF(2^k)$. Finding "good" curves for elliptic curve cryptography would then be easier. This could result in a wider choice of curves than in the case $p = 2$. This method can be extended to finite fields of the form $GF(2^{nm})$, where $2^n > 2m$. In this case $p = 2^n$ is no longer a prime number which forces us to choose the values of E and E' in $GF(2^n)^*$. Fields of this form can also be useful for the recent tripartite Diffie-Hellamn key exchange algorithm [8] or the short signature scheme [3] which require an efficient arithmetic over $GF(p^{kl})$, where $6 < k \leqslant 15$ and l is a prime number greater than 160.

Acknowledgements

The authors would like to thank the anonymous reviewers for their very useful comments. This work has been supported by an *ACI cryptologie 2002* grant from the French ministry of education and research.

References

[1] D. Bailey and C. Paar. Optimal extension fields for fast arithmetic in public-key algorithms. In H. Krawczyk, editor, *Advances in Cryptography – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science (LNCS)*, pages 472–485. Springer-Verlag, 1998.

[2] D. Bailey and C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[3] D. Boneh, H. Shacham, and B. Lynn. Short signatures from the Weil pairing. In *proceedings of Asiacrypt'01*, volume 2139 of *Lecture Notes in Computer Science*, pages 514–532. Springer-Verlag, 2001.

[4] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

[5] R. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent number 5159632, 1992.

[6] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[7] A. Halbutoğullari and Ç. K. Koç. Parallel multiplication in GF(2^k) using polynomial residue arithmetic. *Designs, Codes and Cryptography*, 20(2):155–173, June 2000.

[8] A. Joux. A one round protocol for tripartite Diffie-Hellman. In *4th International Algorithmic Number Theory Symposium (ANTS-IV*, volume 1838 of *Lecture Notes in Computer Science*, pages 385–393. Springer-Verlag, July 2000.

[9] G. A. Jullien, W. Luo, and N. Wigley. High Throughput VLSI DSP Using Replicated Finite Rings. *Journal of VLSI Signal Processing*, 14(2):207–220, November 1996.

[10] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1997.

[11] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[12] N. Koblitz. *A Course in Number Theory and Cryptography*, volume 114 of *Graduate texts in mathematics*. Springer-Verlag, second edition, 1994.

[13] N. Koblitz. *Algebraic aspects of cryptography*, volume 3 of *Algorithms and computation in mathematics*. Springer-Verlag, 1998.

[14] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1997.

[15] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[16] C. Paar, P. Fleischmann, and P. Roelse. Efficient multiplier architectures for galois fields GF(2^{4n}). *IEEE Transactions on Computers*, 47(2):162–170, February 1998.

[17] A. Skavantzos and P. B. Rao. New multipliers modulo $2^N - 1$. *IEEE Transactions on Computers*, 41(8):957–961, August 1992.

[18] N. P. Smart. A comparison of different finite fields for use in elliptic curve cryptosystems. Research report CSTR-00-007, University of Bristol, June 2000.

[19] N. Takagi and S. Yajima. Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem. *IEEE Transactions on Computers*, 41(7):887–891, July 1992.

[20] F. J. Taylor. Large moduli multipliers for signal processing. *IEEE Transactions on Circuits and Systems*, C-28:731–736, Jul 1981.

[21] J. Von Zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[22] Z. Wang, G. A. Jullien, and W. C. Miller. An Efficient Tree Architecture for Modulo $2n + 1$ Multiplication. *Journal of VLSI Signal Processing*, 14(3):241–248, December 1996.

[23] T. Yanik, E. Savaş, and Ç. K. Koç. Incomplete reduction in modular arithmetic. *IEE Proceedings: Computers and Digital Technique*, 149(2):46–52, March 2002.