Parallel Montgomery Multiplication in $GF(2^k)$ using Trinomial Residue Arithmetic

Jean-Claude Bajard[1], Laurent Imbert[1,2], and Graham A. Jullien[2]

[1] LIRMM, CNRS UMR 5506
161 rue Ada, 34392 Montpellier cedex 5, France

[2] ATIPS, CISaC, University of Calgary
2500 University drive NW, Calgary, AB, T2N 1N4, Canada

Abstract

We propose the first general multiplication algorithm in $GF(2^k)$ with a subquadratic area complexity of $\mathcal{O}(k^{8/5}) = \mathcal{O}(k^{1.6})$. We represent the elements of $GF(2^k)$ according to $2n$ pairwise prime trinomials, $T_1, \ldots, T_{2n}$, of degree d, such that $nd \geq k$. Our algorithm is based on Montgomery's multiplication applied to the ring formed by the direct product of the n first trinomials.

1 Introduction

Finite fields [1], and especially the extensions of $GF(2)$, are fundamental in coding theory [2, 3], and cryptography [4, 5]. Developing efficient arithmetic operators in $GF(2^k)$ is a real issue for elliptic curve cryptosystems [6, 7], where the degree, k, of the extension is very large.

Among the many solutions proposed in the literature, we find two classes of algorithms: generic algorithms work for any extension fields, and for any reduction polynomials. The most known general methods are an adaptation of Montgomery's multiplication [8] to binary fields [9], and the approach described by E. Mastrovito [10], where the multiplication is expressed as a matrix-vector product. However, the most efficient implementations use features of the extension fields, such as the type of the base [11, 12, 13, 14], or the form of the irreducible polynomial which define the field. In his Ph.D. thesis [10], E. Mastrovito, proved that some kind of trinomials lead to very efficient implementations; this work was further extended to all trinomials [15]. In [16], F. Rodriguez-Henriquez and Ç. K. Koç propose parallel multipliers based on special irreducible pentanomials.

A common characteristic of all those methods is their quadratic area-complexity; the number of gates is in $\mathcal{O}(k^2)$. Implementations using lookup-tables have been proposed in order to reduce the number of gates. In [17], A. Halbutogullari and Ç. K. Koç, present an original method using a polynomial residue arithmetic with lookup-tables. More recently, B. Sunar [18] proposed a general subquadratic algorithm, which best asymptotic bound, $O(k^{\log_2 3})$, is reached when k is a power of $2, 3$, or 5, and when the reduction polynomial has a low Hamming weight, such as a trinomial or a pentanomial. This approach is based on the Chinese Remainder Theorem (CRT) for polynomials, and Winograd's convolution algorithm.

In this paper, we consider a polynomial residue representation, with n, degree-d trinomials, such that $nd \geq k$. Our approach is based on Montgomery's algorithm, where all computations are performed on the residues, and where large lookup tables are not needed. We prove that, for any degree k, and for any reduction polynomial, the asymptotic area-complexity is $\mathcal{O}(k^{8/5}) = \mathcal{O}(k^{1.6})$. Experimental results are presented, which confirm the efficiency of our algorithm for values, k, of cryptographic interest.

We consider the finite field, $GF(2^k)$, defined by an irreducible polynomial P. We also define a set of $2n$, relatively prime trinomials, $(T_1, \ldots, T_{2n})$, with $\deg T_j = d$, for $j = 1, \ldots, 2n$, and such that $nd \geq k$. We note t_j the degree of the intermediate term of each trinomial T_j, such that $T_j(X) = X^d + X^{t_j} + 1$. We easily remark that for $i, j = 1, \ldots, 2n,\ i \neq j$, we have $\gcd(T_i, T_j) = 1$. Thus, an element $A \in GF(2^k)$ can represented by its residues modulo

$(T_1, \ldots, T_{2n})$. We shall denote $(A_1, \ldots, A_{2n})$, the residue representation of A.

2 Montgomery Multiplication in Polynomial Residue Arithmetic

2.1 Montgomery Multiplication for Integers and Polynomials

Let us start with Montgomery's multiplication over integers. Instead of computing $a\,b \bmod n$, Montgomery's algorithm returns $a\,b\,r^{-1} \bmod n$, where r is such that $\gcd(r, n) = 1$. (In practice n is often a prime number, and r can be chosen as a power of 2). In this paper, we shall refer to r as the Montgomery factor. The computation is accomplished in two steps: we first define $q = -a\,b\,n^{-1} \bmod r$, such that $a\,b + q\,n$ is a multiple of r; a division by r, which reduces to right shifts, then gives the result.

The same idea applies for elements (considered as polynomials here) of any finite extension field. See, e.g. [17] in the case of $GF(2^k)$, and [19] for $GF(p^k)$, with $p > 2$. The polynomial, $R(X) = X^k$, is commonly chosen as the Montgomery factor, because the reduction modulo X^k, and the division by X^k, consist in ignoring the terms of order larger than k for the remainder operation, and shifting the polynomial to the right by k places for the division. In order to compute $A\,B\,R^{-1} \bmod P$, we first define $Q = -A\,B\,P^{-1} \bmod R$, and compute $(A\,B + Q\,P)/R$ using k right-shifts. The only difference with the integer algorithm is that the final subtraction is not required at the end.

2.2 Montgomery over Polynomial Residues

We apply the same scheme when the polynomials A, B, and P are given in their residue representation, i.e., by their remainders modulo a set of pairwise prime polynomials. In this paper, we consider a set of n trinomials $(T_1, \ldots, T_n)$. We define the Montgomery constant as

$$\Gamma = \prod_{i=1}^{n} T_i. \tag{1}$$

We shall thus compute $A\,B\,\Gamma^{-1} \bmod P$. However, unlike the integer and polynomial cases mentioned above, it is important to note, that, in the residue representation, it is not possible to evaluate $(A\,B + Q\,P)/\Gamma$ directly, because the inverse of Γ does not exist modulo Γ. We address this problem by using k extra trinomials $(T_{n+1}, \dots, T_{2n})$, where $\gcd(T_i, T_j) = 1$ for $1 \leq i, j \leq 2n$, $i \neq j$; and by computing $(A\,B + Q\,P)$ over those k extra trinomials. Algorithm 1, bellow, returns $R = A\,B\,\Gamma^{-1} \bmod P$ in a residue representation over the set $(T_1, \dots, T_n, T_{n+1}, \dots, T_{2n})$.

Algorithm 1 [MMTR: Montgomery Multiplication over Trinomial Residues]

Precomputed: $3n$ constant matrices $d \times d$ for the multiplications by $P_i^{-1} \bmod T_i$ (in step 2), by $P_{n+i} \bmod T_{n+i}$ (step 4), and by $\Gamma_{n+i}^{-1} \bmod T_{n+i}$ (step 5), for $i = 1, \dots, n$; (Note that with Mastrovito's algorithm for trinomials [15], we only need to store $2d$ coefficients per matrix.)

Input: $6n$ polynomials of degree at most $d-1$: A_i, B_i, P_i, for $i = 1, \dots, 2n$

Output: $2n$ polynomials of degree at most $d-1$: $R_i = A_i\,B_i\,\Gamma^{-1} \bmod P_i$, for $i = 1, \dots, 2n$

```
1: $(C_1, \dots, C_{2n}) \leftarrow (A_1, \dots, A_{2n}) \times (B_1, \dots, B_{2n})$
2: $(Q_1, \dots, Q_n) \leftarrow (C_1, \dots, C_n) \times (P_1^{-1}, \dots, P_n^{-1})$
3: Newton's interpolation: $(Q_1, \dots, Q_n) \rightsquigarrow (Q_{n+1}, \dots, Q_{2n})$
4: $(R_{n+1}, \dots, R_{2n}) \leftarrow (C_{n+1}, \dots, C_{2n}) + (Q_{n+1}, \dots, Q_{2n}) \times (P_{n+1}, \dots, P_{2n})$
5: $(R_{n+1}, \dots, R_{2n}) \leftarrow (R_{n+1}, \dots, R_{2n}) \times (\Gamma_{n+1}^{-1}, \dots, \Gamma_{2n}^{-1})$
6: Newton's interpolation: $(R_{n+1}, \dots, R_{2n}) \rightsquigarrow (R_1, \dots, R_n)$
```

As in the polynomial case, the final subtraction is not necessary. This can be proved by showing that the polynomial R is fully reduced, i.e., its degree is always less than $k-1$. We note that computing over the set of trinomials, $(T_1, \dots, T_n)$, is equivalent as computing modulo Γ, with $\deg \Gamma = nd$. Given $A, B \in GF(2^k)$, we have $\deg C \leq 2k-2$, and $\deg Q \leq$

$nd-1$; and since $2k-2 \leq nd-1+k$, we get $R=(C+QP)\Gamma^{-1}$ of degree at most $k-1$. In steps 3 and 6, we also remark that two base extensions are required. Most of the complexity of Algorithm 1 depends on those two steps. We give details in the next section.

3 Base Extensions using Trinomial Residue Arithmetic

In this section, we focus on the residue extensions in steps 3 and 6 of Algorithm 1. We shall only consider the extension of Q, from its residues representation $(Q_1,\ldots,Q_n)$ over the set $(T_1,\ldots,T_n)$, to its representation $(Q_{n+1},\ldots,Q_{2n})$, over $(T_{n+1},\ldots,T_{2n})$.[1] Note that this operation is nothing more than an interpolation. We begin this section with a brief recall of an algorithm based on the Chinese Remainder Theorem (CRT), previously used in [18, 17]. Then we focus on the complexity of Newton's interpolation method with trinomials, which, as we shall see further, has a lower complexity.

For the CRT-based interpolation algorithm, we define, $\rho_{i,j}=\left(\frac{\Gamma}{T_j}\right) \bmod T_{n+i}$, and $\nu_i=\left(\frac{\Gamma}{T_i}\right)^{-1} \bmod T_i$, for $i,j=1,\ldots,n$, with Γ defined in (1). Given $(Q_1,\ldots,Q_n)$, we obtain $(Q_{n+1},\ldots,Q_{2n})$ using the Chinese Remainder Theorem for polynomials. We compute $\alpha_i = Q_i\,\nu_i \bmod T_i$, for $i=1,\ldots,n$, and we evaluate

$$Q_{n+j}=\sum_{j=i}^{n}\alpha_i\,\rho_{i,j} \bmod T_{n+j}, \quad \forall j=1,\ldots,n. \tag{2}$$

The evaluation of the α_is is equivalent to n polynomial multiplications modulo T_i. The terms, $\alpha_i\,\rho_{i,j}$, in (2) can be expressed as a matrix-vector product, $Q=Z\alpha$, where Z is a precomputed $n\times n$ matrix. Thus, the CRT-based interpolation requires (n^2+n) modular multiplications modulo a trinomial of degree d, and the precomputation of (n^2+n) matrices $d\times d$. When we do not have any clue about the coefficients of the matrices, an upper-bound for the cost of one polynomial modular multiplication in $GF(2^k)$, is d^2 AND, and $d(d-1)$ XOR, with a latency of $T_A+\lceil\log_2(d)\rceil\,T_X$, where T_A, and T_X, represent the delay for one AND gate, and one XOR gate respectively.

---

[1] The same analysis applies for the reverse operation in step 6.

A second method, that we shall discuss more deeply here, uses Newton's interpolation algorithm. In this approach we first construct an intermediate vector, $(\zeta_1, \ldots, \zeta_n)$ – equivalent to the mixed radix representation for integers – where the ζ_is are polynomials of degree less than d. The vector $(\zeta_1, \ldots, \zeta_n)$ is obtained by the following computations:

$$\begin{cases} \zeta_1 = Q_1 \\ \zeta_2 = (Q_2 + \zeta_1)\ T_1^{-1} \bmod T_2 \\ \zeta_3 = \big((Q_3 + \zeta_1)\ T_1^{-1} - \zeta_2\big)\ T_2^{-1} \bmod T_3 \\ \ \vdots \\ \zeta_n = \big(\ldots\big((Q_n + \zeta_1)\ T_1^{-1} + \zeta_2\big)\ T_2^{-1} + \cdots + \zeta_{n-1}\big)\ T_{n-1}^{-1} \bmod T_n. \end{cases} \tag{3}$$

We then evaluate the polynomials Q_{n+i}, for $i = 1, \ldots, n$, with Horner's rule, as

$$Q_{n+i} = (\ldots((\zeta_n\, T_{n-1} + \zeta_{n-1})\ T_{n-2} + \cdots + \zeta_3)\ T_2 + \zeta_2)\ T_1 + \zeta_1 \bmod T_{n+i}. \tag{4}$$

Algorithm 2, bellow, summarizes the computations.

Algorithm 2 [Newton Interpolation]

Input: $(Q_1, \ldots, Q_n)$

Output: $(Q_{n+1}, \ldots, Q_{2n})$

```
$\zeta_1 \leftarrow Q_1$
for $i = 2, \ldots, n$, in parallel, do
    $\zeta_i \leftarrow Q_i$
    for $j = 1$ to $i - 1$ do
        $\zeta_i \leftarrow \big((\zeta_i + \zeta_j) \times T_j^{-1}\big) \bmod T_i$
for $i = 1, \ldots, n$, in parallel, do
    $Q_{n+i} \leftarrow \zeta_n \bmod T_{n+i}$
    for $j = n - 1$ to $1$ do
        $Q_{n+i} \leftarrow (Q_{n+i} \times T_j + \zeta_j) \bmod T_{n+i}$
```

3.1 Computation of the ζ_is

We remark that the main operation involved in the first part of Algorithm 2 (steps 2 to 5), consists in a modular multiplication of a polynomial of the form $F = (\zeta_i + \zeta_j)$ by the inverse of a trinomial T_j, modulo another trinomial T_i. Since $(T_i, T_j) = 1$, we can use Montgomery multiplication, with T_j playing the role of the Montgomery factor (cf. Section 2.1), to compute

$$\psi = F \times T_j^{-1} \bmod T_i. \tag{5}$$

Let us define $B_{j,i} = T_j \bmod T_i$, such that $B_{j,i}(X) = X^{t_i} + X^{t_j}$. (Note that $B_{j,i} = B_{i,j}$). Clearly, we have $T_j^{-1} \equiv B_{j,i}^{-1} \pmod{T_i}$. Thus, (5) is equivalent to

$$\psi = F \times B_{j,i}^{-1} \bmod T_i. \tag{6}$$

We evaluate (6) as follows: We first compute $\phi = F \times T_i^{-1} \bmod B_{j,i}$, such that $F + \phi \times T_i$ is a multiple of $B_{j,i}$. Thus, $\psi = (F + \phi\, T_i)/B_{j,i}$, is obtained with a division by $B_{j,i}$.

By looking more closely at the polynomials involved in the computations, we remark that $B_{j,i}(X) = X^{t_j}(X^{t_i - t_j} + 1)$, if $t_j < t_i$. (If $t_i < t_j$, we shall consider $B_{j,i}(X) = X^{t_i}(X^{t_j - t_i} + 1)$). In order to evaluate (6), we thus have to compute an expression of the form $F \times \left(X^a \left(X^b + 1\right)\right)^{-1} \bmod T_i$, which can be decomposed into

$$\psi = \left(F \times (X^a)^{-1} \bmod T_i\right) \times \left(X^b + 1\right)^{-1} \bmod T_i. \tag{7}$$

Again, using Montgomery's reduction, with X^a playing the role of the Montgomery factor,[2] we evaluate $F \times (X^a)^{-1} \bmod T_i$ in two steps:

$$\phi = F \times T_i^{-1} \bmod X^a \tag{8}$$

$$\psi = \left(F + \phi \times T_i\right)/X^a \tag{9}$$

Since a is equal to the smallest value between t_i and t_j, we have $a \leq t_i$, and thus $T_i \bmod X^a = T_i^{-1} \bmod X^a = 1$. Hence, (8) rewrites $\phi = F \bmod X^a$, which reduces to the

---

[2] It is easy to see that $\gcd(X^a, T_i) = 1$ always.

truncation of the coefficients of F of order greater than $a-1$. For (9), we first deduce $\phi T_i = \phi X^d + \phi X^{t_i} + \phi$. Since $\deg \phi < a \le t_i < \frac{d}{2}$, there is no recovering between the three parts of ϕT_i, and thus, no operation is required, as shown in Figure 1, where the grey areas represent the a coefficients of ϕ, whereas the white ones represents zeros.

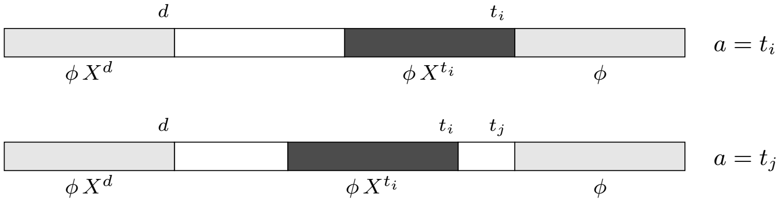

Figure 1: The structure of ϕT_i in both cases $a = t_i$, and $a = t_j$, with the a coefficients to add with F in dark grey

Since the a coefficients of $(F + \phi T_i)$, of order less than a, are thrown away in the division by X^a, we only need to perform the addition with F for the a coefficients which correspond to ϕX^{t_i} (in dark grey in Figure 1). Thus, the operation $F + \phi \times T_i$ reduces to at most a XOR, with a latency T_X of one XOR. The final division by X^a is a truncation, performed at no cost.

Let us now consider the second half of equation (7), i.e., the evaluation of an expression of the form $F \times (X^b + 1)^{-1} \bmod T_i$. (We can notice that F is equal to the value ψ in (9), just computed, and that it has degree at most $d-1$.) Let us consider four steps:

$$F = F \bmod (X^b + 1) \tag{10}$$

$$\phi = F \times T_i^{-1} \bmod (X^b + 1) \tag{11}$$

$$\rho = F + \phi \times T_i \tag{12}$$

$$\psi = \rho / (X^b + 1) \tag{13}$$

For (10), we consider the representation of F in radix X^b; i.e., $F = \sum_{i=0}^{\left\lfloor \frac{d-1}{b} \right\rfloor} F_i \left(X^b\right)^i$. Thus, using the congruence $X^b \equiv 1 \pmod{X^b+1}$, we compute

$$F \bmod (X^b+1) = \sum_{i=0}^{\left\lfloor \frac{d-1}{b} \right\rfloor} F_i,$$

with $(d-b)$ XOR and a latency of $\lceil \log_2((d-1)/b) \rceil T_X$.[3]

The second step, in (11), is a multiplication of two polynomials of degree $b-1$, modulo X^b+1. We first perform the polynomial product $F \times T_i^{-1}$, where T_i^{-1} is precomputed, and we reduce the result using the congruence $X^b \equiv 1 \bmod (X^b+1)$. The cost is thus b^2 AND, and $(b-1)^2$ XOR for the polynomial product, plus $b-1$ XOR for the reduction modulo (X^b+1); a total of $b(b-1)$ XOR. [4] The latency is equal to $T_A + \lceil \log_2(b) \rceil T_X$.

For (12), we recall that b is equal to the positive difference between the t_i and t_j. Thus, we do not know whether $b \le t_i$ or $b > t_i$. In the first case, there is no recovering between the parts of $\phi T_i = \phi X^d + \phi X^{t_i} + \phi$; and ϕX^d is deduced without operation (cf. Figure 2). Thus, $\rho = F + \phi T_i$, only requires $2b$ XOR. If $b > t_i$, however, ϕ and ϕX^{t_i} have $b - t_i$ coefficients in common, as shown in Figure 2. The expression $\rho = F + \phi T_i$ is thus computed with $t_i + 2(b-t_i) + (b + t_i - b) = 2b$ XOR. Thus, in both cases, (12) is evaluated with $2b$ XOR, and with a latency of at most $2T_X$. (T_X only, if $b \le t_i$.)

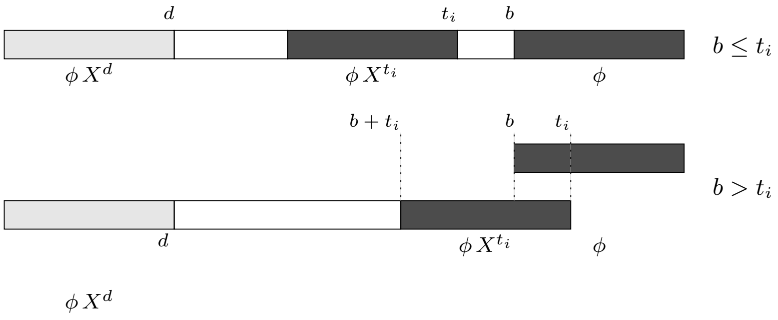

Figure 2: The structure of ϕT_i in both cases $b \le t_i$, and $b > t_i$, and the $2b$ coefficients to add with F in dark grey

---

[3] For $d-1 > b$, we have $\lceil \log_2 \lceil (d-1)/b \rceil \rceil = \lceil \log_2((d-1)/b) \rceil$.

[4] The cost is equivalent as a matrix-vector product using Mastrovito's algorithm, because the construction of the folded matrix is free for X^b+1.

For the last step, the evaluation of ψ in (13), is an exact division; ρ, which is a multiple of X^b+1, has to be divided by X^b+1. This is equivalent to defining ψ such that $\rho = \psi\, X^b + \psi$. As previously, we express ρ and ψ in radix X^b. We have

$$\rho = \sum_{i=0}^{\left\lfloor \frac{d-1}{b} \right\rfloor + 1} \rho_i \left(X^b\right)^i, \qquad \psi = \sum_{i=0}^{\left\lfloor \frac{d-1}{b} \right\rfloor} \psi_i \left(X^b\right)^i.$$

We remark that defining the coefficients of ψ, of order less than b, and greater or equal to $\left(\left\lfloor \frac{d-1}{b} \right\rfloor\right) b$, shown in grey in Figure 3, is accomplished without operation. We have $\psi_0 = \rho_0$, and $\psi_{\left\lfloor \frac{d-1}{b} \right\rfloor} = \rho_{\left\lfloor \frac{d-1}{b} \right\rfloor + 1}$. For the middle coefficients, (i.e., for i from 1 to $\left\lfloor \frac{d-1}{b} \right\rfloor - 1$), we use the recurrence $\psi_i = \rho_i + \psi_{i-1}$.

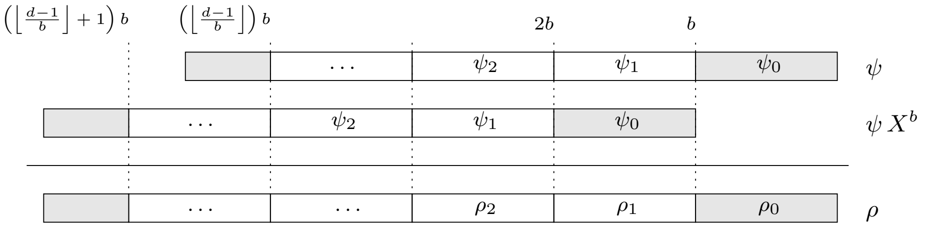

Figure 3: The representations of ρ and ψ in radix X^b

Evaluating (13) thus required $(d-2b)$ XOR, and a latency of $\lceil (d-1)/2b \rceil\, T_X$, taking into account that we start the recurrence, $\psi_i = \rho_i + \psi_{i-1}$, from the two extrema simultaneously.

In Table 1, we recapitulate the computation of $\psi = F \times T_j^{-1} \bmod T_i$ in (5), and its complexity in both the number of binary operations, and time. The total time complexity is equal to

$$T = T_A + \left(4 + \lceil \log_2((d-1)/b) \rceil + \lceil \log_2(b) \rceil + \lceil (d-1)/2b \rceil\right) T_X. \tag{14}$$

So far, the quantities given in Table 1, depend on a and b. In order to evaluate the global complexity for the evaluation of all the ζ_is, me must make assumptions on the t_js, to define more precisely the parameters a, b. In Section 4, we shall give the total cost of (3) when the t_js are equally spaced, consecutive integers.

Equation	# AND	# XOR	Time
(8)	-	-	-
(9)	-	a	T_X
(10)	-	$d-b$	$\lceil \log_2((d-1)/b) \rceil \, T_X$
(11)	b^2	b^2-b+1	$T_A + \lceil \log_2(b) \rceil \, T_X$
(12)	-	$2b$	$2\,T_X$
(13)	-	$d-2b$	$\lceil (d-1)/2b \rceil \, T_X$
Total	b^2	$a+2d+(b-1)^2$	cf. (14)

Table 1: Number of binary operations, and time complexity for $\psi = F \times T_j^{-1} \bmod T_i$

3.2 Computation of the Q_{n+i}s using Horner's rule

When the evaluation of $(\zeta_1, \ldots, \zeta_n)$ is completed, we compute the Q_{n+i}s with the Horner's rule. For $i = 1, \ldots, n$, we have

$$Q_{n+i} = (\ldots((\zeta_n \, T_{n-1} + \zeta_{n-1}) \; T_{n-2} + \cdots + \zeta_3) \; T_2 + \zeta_2) \; T_1 + \zeta_1 \bmod T_{n+i}. \tag{15}$$

In (15), we remark that the main operation is a multiplication of the form $F \times T_j$ mod T_{n+i}, where F is of degree $d-1$, and both T_j, and T_{n+i} are trinomials of degree d. This operation can be expressed as a matrix-vector product, MF, where M is a $(2d+1) \times (d+1)$ matrix composed of the coefficients of T_j. A multiplier architecture was proposed by E. Mastrovito [20], which reduces this matrix M to a $d \times d$ matrix, Z, using the congruences $X^{d+\alpha} \equiv X^{t_{n+i}+\alpha} + X^{\alpha}$, for $\alpha = 0, \ldots, d+1$. The resulting matrix, Z, is usually called the folded matrix, because the $d+1$ last rows of M fall back on the d first ones.

According to our notation, we have, $T_j \bmod T_{n+i} = X^{t_j} + X^{t_{n+i}} = B_{j,n+i}$, for all i, j. Thus, we have to fold a matrix composed of only two non-null coefficients per column, as shown in Figure 4. We remark that the folded matrix, Z, (on the right in Figure 4), is very sparse. By looking more closely, the congruences

$$X^{d+t_j-1} \equiv X^{t_{n+i}+t_j-1} + X^{t_j-1} \pmod{T_{n+i}},$$

$$X^{d+t_{n+i}-1} \equiv X^{2t_{n+i}-1} + X^{t_{n+i}-1} \pmod{T_{n+i}},$$

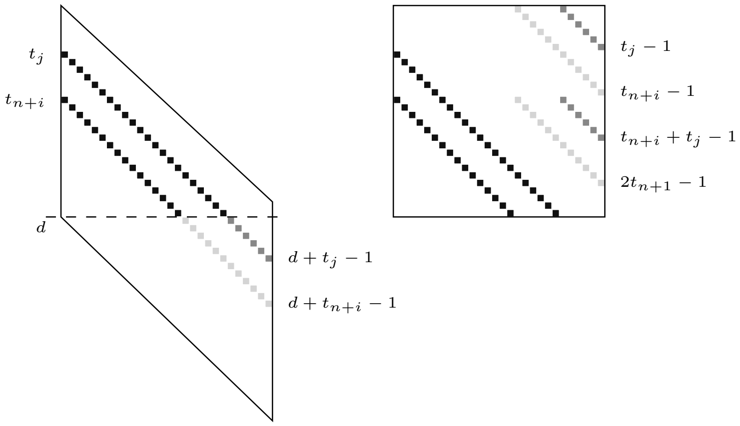

Figure 4: The structures of the unfolded and folded multiplication matrices, for $B_{j,n+i}$ mod T_{n+i}

tell us that, choosing $t_i < d/2$, for $i = 1, \ldots, 2n$, yields $t_j + t_{n+i} - 1 < d$, and $2t_{n+i} - 1 < d$; and thus every coefficients only need to be reduced once. Moreover, we also notice that the matrix, Z, has two non-null coefficients from column 0 to column $d - t_{n+i} - 1$; three from column $d - t_{n+i}$ to column $d - t_j - 1$; and four from column $d - t_j$ to $d - 1$. Thus, it has exactly $2d + t_j + t_{n+i}$ non-null coefficients. Since $t_j, t_{n+i} < d/2$, we can consider that the number of non-zero coefficients is less than $3d$. We study the global complexity of (15), in Section 4.

4 Analysis of the Algorithms

In order to evaluate precisely the cost of Algorithm 1, we consider equally spaced, consecutive t_is, with $t_{i+1} - t_i = r$. Hence, if $j < i$ (as in steps 2 to 5 of Algorithm 1), then $t_j < t_i$, and we have

$$a = t_1 + (j - 1)r, \qquad b = (i - j)r. \tag{16}$$

4.1 Complexity analysis for the computation of the ζ_is

For the first part of the algorithm, i.e., the evaluation of the ζ_is, we remark (cf. Algorithm 2) that, for all i, j, we perform one addition, $(\zeta_i + \zeta_j)$ with polynomials of degree $< d$, followed by one multiplication by T_j^{-1} modulo T_i, which complexity is given in Table 1. Using (16), the following formulas hold:

$$\begin{aligned} \# AND: &\quad \sum_{i=2}^{n} \sum_{j=1}^{i-1} ((i-j)\, r)^2, \\ \# XOR: &\quad \sum_{i=2}^{n} \sum_{j=1}^{i-1} \left(d + (t_1 + (j-1)r) + 2d + ((i-j)r - 1)^2\right), \end{aligned}$$

which, after simplifications, gives

$$\# AND: \quad \frac{r^2 n^2 (n-1)(n+1)}{12}, \tag{17}$$

$$\# XOR: \quad \frac{n(n-1)(r^2n^2 + r^2n - 2rn - 8r + 18d + 6t_1 + 6)}{12}. \tag{18}$$

For the latency, we remark that the polynomials ζ_is, can be computed in parallel, for $i = 1, \ldots, n$, but, that the sum for $j = 1, \ldots, n-1$ (evaluated in steps 4 and 5 of Algorithm 2), is sequential. We also notice that, for a given i, the evaluation of ζ_i can not be completed before we know the previous polynomial ζ_{i-1}. The delay is thus equal to the time required for the addition of ζ_{i-1}, plus the time for the computation of $F \times T_{i-1}^{-1} \bmod T_i$, i.e., when $b = r$. (Remember that r is the difference between two consecutive t_is.) We conclude that the total time complexity for (3) is equal to

$$(n-1)\, T_A + (n-1)\Big(5 + \lceil \log_2((d-1)/r)) \rceil + \lceil \log_2(r) \rceil + \lceil (d-1)/2r \rceil \Big) T_X. \tag{19}$$

For the second Newton's interpolation (step 6 of Algorithm 1), we observe that defining $t_{n+i} = t_{n+1} + (i-1)r$, yields the same complexities. E.g., we can choose $t_1 = 1$, $r = 2$, and $t_{n+1} = 2$. [5]

---

[5] It is also possible to choose $t_1 = 0$. In this case, T_1 is a binomial and we obtain a slightly lower complexity. Also, the condition $2n < d/2$ becomes $2n - 1 < d/2$.

In terms of memory requirements, we have to store polynomials of the form $T_j^{-1}(X) \bmod (X^b+1)$, used to compute (11). How many of them do we need? For a given i, the evaluation of ζ_i, involves $i-1$ polynomials $T_j^{-1}(X) \bmod (X^b+1)$, of degree at most $b-1$, i.e., with b coefficients each. Since b goes from r to $(i-1)r$, we have exactly one polynomial of each degree, ranging from $(r-1)$ to $(i-1)r-1$. The total memory cost, for $i=2,\ldots,n$, is equal to $\sum_{i=2}^{n}\sum_{j=1}^{i-1} j\,r = \dfrac{1}{6}rn(n^2-1)$ bits.

4.2 Complexity Analysis for the Computation of the Q_{n+1}s using Horner's rule

Let us first count the exact number of non-zero coefficients in the folded matrices, Z, given in Section 3.2. With $t_j = t_1 + (j-1)r$, and $t_{n+i} = t_{n+1} + (i-1)r$, defined as above, we get $2d + t_i + t_{n+j} = 2d + t_1 + t_{n+1}(i+j-2)r$ non-zero values for each matrix. Thus, the matrix-vector product used to compute the expressions of the form $F \times T_j \bmod T_{n+i}$ requires $2d+t_1+t_{n+1}(i+j-2)r$ AND, and $d+t_1+t_{n+1}+(i+j-2)r$ XOR.[6] Because all the products are performed in parallel, and because each inner-product involves at most 4 values, the latency is equal to $T_A + 2T_X$.

The computation of Q_{n+i} in (15) is sequential. Each iteration performs one matrix-vector product, followed by one addition with a polynomial, ζ_j, (cf. step 9 of Algorithm 2) of degree at most $d-1$. We thus get

$$\begin{aligned}
\#\,AND: &\quad \sum_{j=1}^{n}\sum_{i=1}^{n-1}\left(2d+t_1+t_n+(i+j-2)r\right),\\
\#\,XOR: &\quad \sum_{j=1}^{n}\sum_{i=1}^{n-1}\left(d+t_1+t_n+(i+j-2)r+d\right),
\end{aligned}$$

or equivalently (noticing that the two sums, above, are equal),

$$\#\,AND,\ \#\,XOR: \quad \frac{1}{2}n(n-1)(4d+2rn-3r+2t_1+2t_{n+1}). \tag{20}$$

The total delay for (15) is thus: $(n-1)(T_A+3T_X)$.

---

[6] We have $(d-t_{n+i}) + 2(t_{n+i}-t_j) + 3(t_j) = d + t_j + t_{n+i}$; hence the result.

4.3 Complexity Analysis for Newton's Interpolation

The total complexity for Newton's interpolation is the sum of the complexities obtained for the computation of the ζ_is in Section 4.1, and for evaluation of the Q_{n+i}s with Horner's rule in Section 4.2. We have

$$\# AND = \frac{1}{12}\, n(n-1)(r^2n^2 + 12rn + r^2n + 12t_1 - 18r + 24d + 12t_{n+1}), \tag{21}$$

$$\# XOR = \frac{1}{12}\, n(n-1)(r^2n^2 + 10rn + r^2n + 18t_1 - 26r + 42d + 12t_{n+1} + 6), \tag{22}$$

with a latency of

$$2(n-1)\, T_A + (n-1)\Big(8 + \lceil \log_2((d-1)/r)\rceil + \lceil \log_2(r)\rceil + \lceil (d-1)/2r\rceil\Big) T_X, \tag{23}$$

or equivalently

$$2(n-1)\, T_A + \mathcal{O}(n + \frac{nd}{r})T_X.$$

4.4 Complexity Analysis of MMTR

In Algorithm 1, we note that steps 1, 2, 4, and 5 are accomplished in parallel. In step 1, we perform $2n$ multiplications of the form, $A_i \times B_i \bmod T_i$. Using Mastrovito's algorithm for trinomials [15], it requires d^2 AND, and d^2-1 XOR; thus the cost of step 1 is $2nd^2$ AND, and $2n(d^2-1)$ XOR. In steps 2, 4, and 5, we perform $3n$ constant multiplications, expressed as $3n$ matrix-vector products of the form ZF, where Z is a $d \times d$ precomputed matrix[7]; the complexity is $3nd^2$ AND, and $3nd(d-1)$ XOR. Not forgetting to consider the n additions in step 4, the complexity for steps 1, 2, 4, and 5 is: $5nd^2$ AND, and $5nd^2 - 2nd - 2n$ XOR, with a latency of $4\,T_A + (1 + 4\,\lceil \log_2(d)\rceil)\, T_X$.

We obtain the total complexity of Algorithm 1 by adding the complexity formulas for steps 1, 2, 4, and 5, plus the cost of two Newton's interpolation. The number of gates is:

$$\begin{aligned} \# AND: \quad & \frac{1}{6}n\Big(r^2n^3 + 12rn^2 + 12\,(t_1 + t_{n+1} + 2d)\,(n-1) \\ & \quad - 30rn - r^2n + 30d^2 + 18r\Big), \end{aligned} \tag{24}$$

---

[7] We only need to store $2d$ values.

$$\# XOR: \quad \frac{1}{6}n\Big(r^2n^3 + 10rn^2 + 6n + 6\left(2t_{n+1} + 3t_1\right)(n-1) \\ + 42dn - 36rn - r^2n + 30d^2 - 54d - 18 + 26r\Big); \tag{25}$$

and the delay is equal to

$$4n\,T_A + \Big((n-1)\Big(8 + \lceil\log_2(d-1)/r)\rceil + \lceil\log_2(r)\rceil + \lceil(d-1)/2r\rceil\Big) + 4\lceil\log_2(d)\rceil + 1\Big)T_X, \tag{26}$$

that we express, for simplicity, as

$$4n\,T_A + \mathcal{O}\left(n + \frac{nd}{r}\right)T_X. \tag{27}$$

5 Discussions and Comparisons

The parameters n, d, and t that appear in the complexity formulas above, make the comparison of our algorithm with previous implementations a difficult task. To make it easier, let us assume that $n = k^x$, and $d = k^{1-x}$. Clearly, we have $nd = k$. Since we need $2n$ trinomials of degree less than d, having their intermediate coefficient of order less then $d/2$ (see Section 3.2), the parameters k, x must satisfy $k^{1-2x} > 4$, which is equivalent to $x < \frac{1}{2}$.[8] Thus, in the next AND and XOR counts, we only take into account the terms in k^{2-x}, k^{1+x}, and k^{4x}, and we also consider $t_1 = 0$, $t_{n+1} = n$, and $r = 1$, which seems to be optimal. For the latency, we remark from Table 1, that the time complexity is mostly influenced by the term in $(d-1)/2b$.

Hence, the total complexity for Montgomery multiplication over residues (MMTR) is:

$$\# AND: \quad 5k^{2-x} + 4k^{1+x} + \frac{r^2}{6}k^{4x} + \mathcal{O}(k^{3x}), \tag{28}$$

and

$$\# XOR: \quad 5k^{2-x} + 7k^{1+x} + \frac{r^2}{6}k^{4x} + \mathcal{O}(k^{3x}), \tag{29}$$

for a latency of

$$4k^x\,T_A + \mathcal{O}(k)\,T_X. \tag{30}$$

---

[8] We have $x < (1 - \log_k(4))/2$, and $\lim\limits_{k\to+\infty} = \lim\limits_{k\to 0}\log_k(4) = 0$.

In the literature, the area complexity is given according to the number of XOR gates. Most of the studies are dedicated to specific cases, where the irreducible polynomials used to define the field, are trinomials [15], or special pentanomials [16], of the form $X^k + X^{t+1} + X^t + X^{t-1} + 1$. In table 2, we compare our algorithm with Montgomery's multiplication [9], the trinomial, and pentanomial approaches. We give the number of XOR gates, for extensions of degrees, $k = nd$, ranging from 163 to 2068 bits. For each example, we also consider the costs for the largest prime, p, less than $k = nd$. We note that, $k = 196 = 7 \times 28$, is the smallest integer for which our algorithm has the fewer number of XOR gates. It is important to note that algorithm MMTR is mainly to be compared with Montgomery, since they are both general algorithms, that do not require a special form for the irreducible polynomial which defines the finite field. However, we remark that for four of the primes numbers recommended by the NIST for elliptic curve cryptography over $GF(2^k)$, i.e., $p = 233$, $p = 283$, $p = 409$, $p = 571$, our solution is cheaper than the trinomial and pentanomial algorithms.[9]

Whereas, the three other methods have an area cost in $\mathcal{O}(k^2)$, the asymptotic complexity of our algorithm, reached for $x = 2/5$, is in $\mathcal{O}(k^{1.6})$. For completeness, we give the exact complexity formula:

$$\frac{31}{6}k^{8/5} + 7k^{7/5} + \frac{11}{3}k^{6/5} - 9k - \frac{43}{6}k^{4/5} + \frac{4}{3}k^{2/5}. \tag{31}$$

6 Conclusions

We proposed a modular multiplication algorithm over finite extension fields, $GF(2^k)$, with an area complexity of $\mathcal{O}(k^{1.6})$. Our experimental results confirm its efficiency for extensions of large degree, of great interest for elliptic curve cryptography. For such applications, a major advantage of our solution, is that it allows the use of extension fields for which an irreducible trinomial or special pentanomial, such as in [16], does not exist.

---

[9] The reduction polynomials recommended by the NIST are all trinomials or pentanomials.

Parameters	Montgomery [9]	Pentanomials [16]	Trinomials [15]	MMTR
$k = 165\ (5, 33)$	54,615	27,552	27,224	31,955
$p = 163$	53,301	26,892	26,568	31,955
$k = 196\ (7, 28)$	77,028	38,805	38,415	36,743
$p = 193$	74,691	37,632	37,248	36,743
$k = 238\ (7, 34)$	113,526	57,117	56,643	51,443
$p = 233$	108,811	54,752	54,288	51,443
$k = 288\ (8, 36)$	166,176	83,517	82,943	67,712
$p = 283$	160,461	80,652	80,088	67,712
$k = 414\ (9, 46)$	343,206	172,221	171,395	121,098
$p = 409$	334,971	168,096	167,280	121,098
$k = 572\ (11, 52)$	654,940	328,325	327,183	194,689
$p = 571$	652,653	327,180	326,040	194,689
$k = 1024\ (16, 64)$	2,098,176	1,050,621	1,048,575	459,200
$p = 1021$	2,085,903	1,044,480	1,042,440	459,200
$k = 2068\ (22, 94)$	8,555,316	4,280,757	4,276,623	1,351,548
$p = 2063$	8,514,001	4,260,092	4,255,968	1,351,548

Table 2: XOR counts for our Montgomery multiplication over trinomial residue arithmetic (MMTR), compared to other best known methods

References

[1] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications.* Cambridge University Press, Cambridge, England, revised edition, 1994.

[2] R. W. Hamming. *Coding and information theory.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

[3] M. Sudan. Coding theory: Tutorial and survey. In *Proceedings of the 42th Annual Symposium on Fundations of Computer Science – FOCS 2001*, pages 36–53. IEEE Computer Society, 2001.

[4] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography.* CRC Press, 1997.

[5] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.

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

[7] V. S. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology – CRYPTO '85*, volume 218 of *LNCS*, pages 417–428. Springer-Verlag, 1986.

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

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

[10] E. D. Mastrovito. *VLSI Architectures for Computations in Galois Fields.* PhD thesis, Linköping University, Linköping, Sweden, 1991.

[11] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics*, 22(2):149–161, 1988–1989.

[12] S. T. J. Fenn, M. Benaissa, and D. Taylor. $GF(2^m)$ multiplication and division over dual basis. *IEEE Transactions on Computers*, 45(3):319–327, March 1996.

[13] H. Wu, M. A. Hasan, and I. F. Blake. New low-complexity bit-parallel finite field multipliers using weakly dual bases. *IEEE Transactions on Computers*, 47(11):1223–1234, November 1998.

[14] M. A. Hasan, M. Z. Wang, and V. K. Bhargava. A modified Massey-Omura parallel multiplier for a class of finite field. *IEEE Transactions on Computers*, 42(10):1278–1280, October 1993.

[15] B. Sunar and Ç. K. Koç. Mastrovito multiplier for all trinomials. *IEEE Transactions on Computers*, 48(5):522–527, May 1999.

[16] F. Rodriguez-Henriquez and Ç. K. Koç. Parallel multipliers based on special irreducible pentanomials. *IEEE Transactions on Computers*, 52(12):1535–1542, December 2003.

[17] 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.

[18] B. Sunar. A generalized method for constructing subquadratic complexity $GF(2^k)$ multipliers. *IEEE Transactions on Computers*, 53(9):1097–1105, September 2004.

[19] J.-C. Bajard, L. Imbert, C. Nègre, and T. Plantard. Multiplication in $GF(p^k)$ for elliptic curve cryptography. In *Proceedings 16th IEEE symposium on Computer Arithmetic – ARITH 16*, pages 181–187, 2003.

[20] E. D. Mastrovito. VLSI designs for multiplication over finite fields $GF(2^m)$. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes – AAECC-6*, volume 357 of *LNCS*, pages 297–309. Springer-Verlag, 1989.