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

Research Report LIRMM-04040

October 2004

Abstract

We propose the first general multiplication algorithm in GF(2^k) with a subquadratic area complexity of $\mathcal{O}(k^{8/5}) = \mathcal{O}(k^{1.6})$. Using the Chinese Remainder Theorem, we represent the elements of GF(2^k); i.e. the polynomials in GF(2)[X] of degree at most $k-1$, by their remainder modulo a set of n pairwise prime trinomials, $T_1, \dots, T_n$, of degree d and such that $nd \geq k$. Our algorithm is based on Montgomery's multiplication applied to the ring formed by the direct product of the trinomials.

Keywords: Finite field arithmetic, Montgomery multiplication, Polynomial residue arithmetic, Trinomial, Pentanomial, Elliptic curve cryptography

1 Introduction

Finite fields [10], and especially the extensions of GF(2), are fundamental in coding theory [6, 18], and cryptography [13, 7]. Developing efficient arithmetic operators in GF(2^k) is a real issue for elliptic curve cryptosystems [9, 14], where the degree, k, of the extension must be large ($160 \leq k \leq 600$).

The solutions proposed in the literature, can be classified into two classes of methods: the generic and specific 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 [15] to binary fields [2], and the approach described by E. Mastrovito [12], where the multiplication is expressed as a matrix-vector product. However, the most efficient implementations are specific algorithms which use features of the extension fields, such as the type of the base [16, 4, 21, 8], or the form of the irreducible polynomial defining the field. In his Ph.D. thesis [12], E. Mastrovito, proved that some kind of trinomials lead to very efficient implementations; this work was further extended to all trinomials [20]. In [17], 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 [5], A. Halbutogullari and Ç. K. Koç, present an original method using a polynomial residue arithmetic with lookup-tables. More recently, B. Sunar [19] proposed a general subquadratic algorithm, whose 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, using 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 extensions 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, $(T_1, \dots, T_{2n})$, with $\deg T_j = d$, for $j = 1, \dots, 2n$, and such that $nd \geq k$. We denote t_j the degree of the intermediate term of each trinomial T_j, such that $T_j(X) = X^d + X^{t_j} + 1$. As we shall see further, we also need $t_j < d/2$ (cf. Section 3). Using the Chinese Remainder Theorem, an element $A \in GF(2^k)$ can be represented by its residues modulo $(T_1, \dots, T_{2n})$. We shall denote $(A_1, \dots, A_{2n})$, the residue representation of A. We give more details in Section 2.2.

2 Montgomery Multiplication in Polynomial Residue Arithmetic

In this section we briefly recall Montgomery's multiplication for integers and polynomials. We then present in more details its generalization to polynomial residue arithmetic.

2.1 Montgomery Multiplication for Integers and Polynomials

Let us start with Montgomery's multiplication over integers [15]. Montgomery's algorithm returns $a\,b\,r^{-1} \bmod n$, where r satisfies $\gcd(r, n) = 1$. (In practice n is almost always an odd number; thus r can be chosen as a power of 2). In this paper, we shall refer to r as the Montgomery factor. The idea is to replace the costly division by n, by a very cheap division by r. 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 a value congruent to $abr^{-1} \bmod n$ and less than $2n$. If it is larger than n, a subtraction by n gives the final result. A vast amount of research have been dedicated to Montgomery's algorithms. E.g., the interested reader can find more details in [3] and [13], chapter 14.

The same idea applies for any finite extension field, $\mathrm{GF}(p^k) \cong \mathrm{GF}(p)[X]/(f)$, where f is a monic irreducible polynomial of degree k in $\mathrm{GF}(p)[X]$. In other words, this means that the elements of $\mathrm{GF}(p^k)$ can be represented as the polynomials of degree at most $k-1$, with coefficients in $\{0, \dots, p-1\}$. See, e.g. [5] in the case of $\mathrm{GF}(2^k)$, and [1] for general extension fields, $\mathrm{GF}(p^k)$, with $p > 2$. The polynomial, $R = X^k$, is commonly chosen as the Montgomery factor, because the reduction modulo X^k, and the division by X^k are simple operations. Indeed, they 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 $ABR^{-1} \bmod P$, we first define $Q = -ABP^{-1} \bmod R$, and compute $(AB + QP)/R$, where the division by R

is performed using k right-shifts. The only difference with the integer case is that the final correction is not necessary at the end because the result is already a polynomial of degree at most $k-1$.

2.2 Montgomery Multiplication over Polynomial Residues

Let $(T_1, \ldots, T_n)$ be a set of n relatively prime trinomials. We define Γ of degree $n \times d \geq k$ as

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

The Chinese Remainder Theorem (CRT), which uses the following ring isomorphism,

$$\begin{array}{ccc} \mathrm{GF}(2)[X]/(\Gamma) & \longrightarrow & \mathrm{GF}(2)[X]/(T_1) \times \cdots \times \mathrm{GF}(2)[X]/(T_n) \\ U & \longmapsto & (U \bmod T_1, \ldots, U \bmod T_n)\,, \end{array} \tag{2}$$

tells us that every polynomial, $U \in \mathrm{GF}(2)[X]$, of degree less than $k \leq nd$, is uniquely represented by its remainders modulo $T_1, \ldots, T_n$.

In the following, for every $A \in \mathrm{GF}(2^k)$ (remember that A can be represented as a polynomial in $\mathrm{GF}(2)[X]$, of degree at most $k-1$), we denote $(A_1, \ldots, A_n)$ its residue representation modulo $(T_1, \ldots, T_n)$, or equivalently modulo Γ. In Algorithm 1 below, we shall also need its residue representation modulo $\Gamma' = \prod_{i=1}^{n} T_{n+i}$, for an extra set of n relativelt prime trinomials $(T_{n+1}, \ldots, T_{2n})$, that we shall refer to as $(A_{n+1}, \ldots, A_{2n})$.

We apply Montgomery's scheme to the polynomials A, B, and P given in their residue representation, i.e., by their remainders modulo $(T_1, \ldots, T_n)$. Note that although P is of degree k, in Algorithm 1 we only need its value modulo Γ'.

In our residue version, Γ also plays the role of the Montgomery factor; i.e., we compute $AB\,\Gamma^{-1} \bmod P$. However, unlike the integer and polynomial cases mentioned above, it is important to note that in the residue representation, $(AB+QP)/\Gamma$ can not be evaluated directly (that is modulo Γ), simply because the inverse of Γ does not exist modulo Γ. We address this problem by using n extra trinomials $(T_{n+1}, \ldots, T_{2n})$, such that $\gcd(T_i, T_j) = 1$ for $1 \leq i, j \leq 2n$, $i \neq j$; and by computing $(AB + QP)$ modulo these n extra trinomials. Algorithm 1, below, returns $R = AB\,\Gamma^{-1} \bmod P$ in the residue representation; i.e. we obtain $(R_1, \ldots, R_{2n})$, the remainders of R modulo $(T_1, \ldots, T_{2n})$, or equivalently modulo $\Gamma \times \Gamma'$.

As in the polynomial case, the final subtraction is not necessary. This can be proven by showing that the polynomial R is fully reduced, i.e., its degree is always less than $k-1$. Indeen,

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 $i = 1, \ldots, n$; (Note that with Mastrovito's algorithm for trinomials [20], we only need to store $2d$ coefficients per matrix.)

Input: $5n$ polynomials of degree at most $d-1$: A_i, B_i, for $i = 1, \ldots, 2n$, and P_{n+i} for $i = 1, \ldots, n$

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

1: $(C_1, \ldots, C_{2n}) \leftarrow (A_1, \ldots, A_{2n}) \times (B_1, \ldots, B_{2n})$

2: $(Q_1, \ldots, Q_n) \leftarrow (C_1, \ldots, C_n) \times (P_1^{-1}, \ldots, P_n^{-1})$

3: Newton's interpolation: $(Q_1, \ldots, Q_n) \rightsquigarrow (Q_{n+1}, \ldots, Q_{2n})$

4: $(R_{n+1}, \ldots, R_{2n}) \leftarrow (C_{n+1}, \ldots, C_{2n}) + (Q_{n+1}, \ldots, Q_{2n}) \times (P_{n+1}, \ldots, P_{2n})$

5: $(R_{n+1}, \ldots, R_{2n}) \leftarrow (R_{n+1}, \ldots, R_{2n}) \times (\Gamma_{n+1}^{-1}, \ldots, \Gamma_{2n}^{-1})$

6: Newton's interpolation: $(R_{n+1}, \ldots, R_{2n}) \rightsquigarrow (R_1, \ldots, R_n)$

given A, B, of degree at most $k-1$, we first compute $C = A \times B$ (Step 1) of degree $\deg C \leq 2k-2$. Then in Step 2, we compute $Q = C \times P^{-1} \bmod \Gamma$, of degree less than the degree of Γ, that is at most $nd-1$. Since $\deg P = k$, we deduce $\deg QP \leq nd - 1 + k$; and since $2k - 2 < nd - 1 + k$, we get $R = (C + QP)\Gamma^{-1}$ of degree at most $(nd - 1 + k) - (nd) = k - 1$.

In steps 3 and 6, we remark that two base extensions (implemented using Newton's interpolation technique) are required. Since all the other steps can be performed in parallel, the complexity of Algorithm 1 mainly depends on these two steps. We analyze them in 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 $(Q_1, \ldots, Q_n)$ modulo Γ, to its representation $(Q_{n+1}, \ldots, Q_{2n})$, modulo Γ'.[1] Note that this operation is nothing else than an interpolation. We begin this section with a brief recall of an algorithm based on the Chinese Remainder Theorem (CRT), previously used in [19, 5]. Then we focus on the complexity of Newton's interpolation method with trinomials, which, as we shall see further, has a lower complexity.

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

For the CRT-based interpolation algorithm, with Γ defined in (1), we denote $\gamma_{i,j} = (\Gamma/T_i) \bmod T_j$, and $\overline{\Gamma_i} = (\Gamma/T_i)^{-1} \bmod T_i$, for $i, j = 1, \ldots, n$. Given $(Q_1, \ldots, Q_n)$, we obtain $(Q_{n+1}, \ldots, Q_{2n})$ using the Chinese Remainder Theorem for polynomials. We compute $\beta_i = Q_i \, \overline{\Gamma_i} \bmod T_i$, for $i = 1, \ldots, n$, and we evaluate

$$Q_{n+j} = \sum_{i=1}^{n} \beta_i \, \gamma_{i,n+j} \bmod T_{n+j}, \quad \forall j = 1, \ldots, n. \tag{3}$$

The evaluation of the β_is is equivalent to n polynomial multiplications modulo T_i. Each term, $\beta_i \, \gamma_{i,n+j}$, in (3) can be expressed as a matrix-vector product, $Q = Z\beta$, 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 such matrix-vector product, 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.

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_i's are polynomials of degree less than d. The vector $(\zeta_1, \ldots, \zeta_n)$ is obtained by the following computations:[2]

$$\begin{cases} \zeta_1 = Q_1 \\ \zeta_2 = (Q_2 + \zeta_1) \, T_1^{-1} \bmod T_2 \\ \zeta_3 = \big((Q_3 + \zeta_1) \, T_1^{-1} + \zeta_2\big) \, T_2^{-1} \bmod T_3 \\ \quad \vdots \\ \zeta_n = \big(\ldots\big((Q_n + \zeta_1) \, T_1^{-1} + \zeta_2\big) \, T_2^{-1} + \cdots + \zeta_{n-1}\big) \, T_{n-1}^{-1} \bmod T_n. \end{cases} \tag{4}$$

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

$$Q_{n+i} = (\ldots((\zeta_n \, T_{n-1} + \zeta_{n-1}) \, T_{n-2} + \cdots + \zeta_3) \, T_2 + \zeta_2) \, T_1 + \zeta_1 \bmod T_{n+i}. \tag{5}$$

Algorithm 2, below, summarizes the computations.

[2] In (4), the additions must be replaced by subtractions if the characteristic of the field is $\neq 2$.

Algorithm 2 [Newton Interpolation]

Input: $(Q_1, \ldots, Q_n)$, the residue representation of Q modulo Γ

Output: $(Q_{n+1}, \ldots, Q_{2n})$, the residue representation of Q modulo Γ'

1: $\zeta_1 \leftarrow Q_1$

2: **for** $i = 2, \ldots, n$, **in parallel**, **do**

3: $\quad \zeta_i \leftarrow Q_i$

4: $\quad$ **for** $j = 1$ to $i - 1$ **do**

5: $\quad\quad \zeta_i \leftarrow \Big((\zeta_i + \zeta_j) \times T_j^{-1} \Big) \bmod T_i$

6: **for** $i = 1, \ldots, n$, **in parallel**, **do**

7: $\quad Q_{n+i} \leftarrow \zeta_n \bmod T_{n+i}$

8: $\quad$ **for** $j = n - 1$ to 1 **do**

9: $\quad\quad Q_{n+i} \leftarrow (Q_{n+i} \times T_j + \zeta_j) \bmod T_{n+i}$

In the following, we analyze very thoroughly the Steps 2 to 5 for the computation of the ζ_i's in section 3.1, and Steps 6 to 9 in Section 3.2 for the evaluation of Q_{n+i} using Horner's rule.

3.1 Computation of the ζ_i's

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

$$V = U \times T_j^{-1} \bmod T_i. \tag{6}$$

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, (6) is equivalent to

$$V = U \times B_{j,i}^{-1} \bmod T_i. \tag{7}$$

We evaluate (7) as follows: We first compute $\mu = U \times T_i^{-1} \bmod B_{j,i}$, such that $U + \mu T_i$ is a multiple of $B_{j,i}$. Thus, $V = (U + \mu T_i)/B_{j,i}$, 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 (7), we thus have to compute an expression of the form $U \times \left(X^a \left(X^b+1\right)\right)^{-1}$ mod T_i, which can be decomposed into

$$V = \left(U \times (X^a)^{-1} \bmod T_i\right) \times \left(X^b + 1\right)^{-1} \bmod T_i. \tag{8}$$

Let us first compute $\phi = \left(U \times (X^a)^{-1} \bmod T_i\right)$. Again, using Montgomery's reduction, with X^a playing the role of the Montgomery factor,[3] we evaluate W in two steps:

$$\rho = U \times T_i^{-1} \bmod X^a \tag{9}$$

$$\phi = (U + \rho\, T_i)\,/X^a \tag{10}$$

Since $a = \min(t_i, t_j)$, we have $a \leq t_i$, and thus $T_i \bmod X^a = T_i^{-1} \bmod X^a = 1$. Hence, (9) rewrites $\rho = U \bmod X^a$, which reduces to the truncation of the coefficients of U of order greater than $a-1$. For (10), we first deduce $\rho\, T_i = \rho\, X^d + \rho\, X^{t_i} + \rho$. Since $\deg \rho < a \leq t_i < \frac{d}{2}$, there is no overlap between the three parts of $\rho\, T_i$, and thus, no operation is required to define $\rho\, T_i$. In Figure 1, the grey areas represent the a coefficients of ρ, whereas the white areas represent zeros.

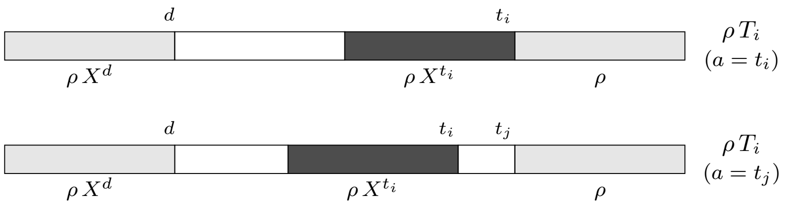

Figure 1: The structure of $\rho\, T_i$ in both cases $a = t_i$, and $a = t_j$, with the a coefficients to add with U in dark grey

Since the a coefficients of $(U + \rho\, T_i)$, of order less than a, are thrown away in the division by X^a, we only need to perform the addition with U for the a coefficients which correspond to $\rho\, X^{t_i}$ (in dark grey in Figure 1). Thus, the operation $U + \rho\, T_i$ 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 (8), i.e., the evaluation of the expression $V = \phi \times (X^b+1)^{-1} \bmod T_i$, where $\phi = U \times (X^a)^{-1} \bmod T_i$ is the polynomial computed in (10).

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

Note that $\deg \phi \le d-1$. Let us consider four steps:

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

$$\psi = \phi \times T_i^{-1} \bmod (X^b+1) \tag{12}$$

$$\omega = \phi + \psi\, T_i \tag{13}$$

$$V = \omega/(X^b+1) \tag{14}$$

For (11), we consider the representation of ϕ in radix X^b; i.e., $\phi = \sum_{i=0}^{\lfloor \frac{d-1}{b} \rfloor} \phi_i \,(X^b)^i$. Thus, using the congruence $X^b \equiv 1 \pmod{X^b+1}$, we compute

$$\phi \bmod (X^b+1) = \sum_{i=0}^{\lfloor \frac{d-1}{b} \rfloor} \phi_i,$$

with $(d-b)$ XOR and a latency of $\lceil \log_2((d-1)/b) \rceil\, T_X$.[4]

The second step, in (12), is a multiplication of two polynomials of degree $b-1$, modulo X^b+1. We first perform the polynomial product $\phi \times T_i^{-1}$, 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.[5] The latency is equal to $T_A + \lceil \log_2(b) \rceil\, T_X$.

For (13), 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 overlapping between the parts of $\psi\, T_i = \psi\, X^d + \psi\, X^{t_i} + \psi$; and $\psi\, T_i$ is deduced without any operation (cf. Figure 2). Thus, $\omega = \phi + \psi\, T_i$, only requires $2b$ XOR. If $b > t_i$, however, ψ and $\psi\, X^{t_i}$ have $b - t_i$ coefficients in common, as shown in Figure 2. The expression $\omega = \phi + \psi\, T_i$ is thus computed with $t_i + 2(b - t_i) + (b + t_i - b) = 2b$ XOR. Thus, in both cases, (13) is evaluated with $2b$ XOR, and with a latency of at most $2\,T_X$. (T_X only, if $b \le t_i$.)

For the last step, the evaluation of V in (14), is an exact division; ω, which is a multiple of X^b+1, has to be divided by X^b+1. This is equivalent to defining α such that $\omega = \alpha\, X^b + \alpha$. As previously, we express ω and α in radix X^b. We have

$$\omega = \sum_{i=0}^{\lfloor \frac{d-1}{b} \rfloor + 1} \omega_i \,(X^b)^i, \qquad \alpha = \sum_{i=0}^{\lfloor \frac{d-1}{b} \rfloor} \alpha_i \,(X^b)^i.$$

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

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

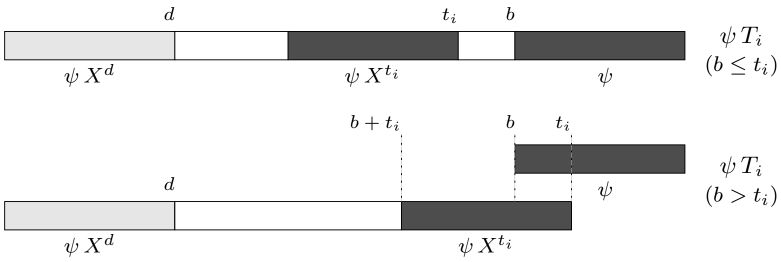

Figure 2: The structure of $\psi\, T_i$ in both cases $b \le t_i$, and $b > t_i$, and the $2b$ coefficients to add with ϕ in dark grey

We remark that defining the coefficients of α, 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 $\alpha_0 = \omega_0$, and $\alpha_{\left\lfloor \frac{d-1}{b} \right\rfloor} = \omega_{\left\lfloor \frac{d-1}{b} \right\rfloor + 1}$. For the middle coefficients, (i.e., for i from 1 to $\left\lfloor \frac{d-1}{b} \right\rfloor - 1$), we use the recurrence $\alpha_i = \omega_i + \alpha_{i-1}$.

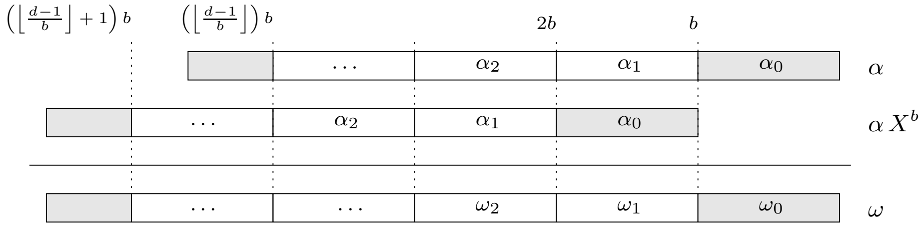

Figure 3: The representations of ω and α in radix X^b

Evaluating (14) thus required $(d-2b)$ XOR, and a latency of $\lceil (d-1)/2b \rceil\, T_X$, taking into account that we start the recurrence, $\alpha_i = \omega_i + \alpha_{i-1}$, from the two extrema simultaneously.

In Table 1, we recapitulate the computation of $V = U \times T_j^{-1} \bmod T_i$ in (6), and its complexity in both the number of binary operations, and time. The total time complexity is equal to

$$T = T_A + (4 + \lceil \log_2((d-1)/b) \rceil + \lceil \log_2(b) \rceil + \lceil (d-1)/2b \rceil)\, T_X. \tag{15}$$

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_i's, me must make assumptions on the t_j's, to define more precisely the parameters a, b. In Section 4, we shall give the total cost of (4) when the t_j's are equally spaced, consecutive integers.

Equation	# AND	# XOR	Time
(9)	-	-	-
(10)	-	a	T_X
(11)	-	$d-b$	$\lceil \log_2((d-1)/b) \rceil \, T_X$
(12)	b^2	b^2-b+1	$T_A + \lceil \log_2(b) \rceil \, T_X$
(13)	-	$2b$	$2\,T_X$
(14)	-	$d-2b$	$\lceil (d-1)/2b \rceil \, T_X$
Total	b^2	$a+2d+(b-1)^2$	cf. (15)

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

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

$$Q_{n+i} = (\ldots((\zeta_n \, T_{n-1} + \zeta_{n-1}) \, T_{n-2} + \cdots + \zeta_3) \, T_2 + \zeta_2) \, T_1 + \zeta_1 \bmod T_{n+i}. \tag{16}$$

In (16), we remark that the main operation is a multiplication of the form $U \times T_j \bmod T_{n+i}$, where U 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, $M \times U$, where M is a $(2d+1) \times (d+1)$ matrix composed of the coefficients of T_j. In $\mathrm{GF}(2^k)$, a straightforward way to accomplish the multiplication operation is to perform a polynomial multiplication, followed by and modular reduction. The polynomial product can be expressed as a matrix-vector product MF, where M is a $(2d+1) \times (d+1)$ matrix composed of the coefficients, τ, of T_j as follows:

$$\begin{pmatrix}
\tau_0 & 0 & 0 & \ldots & 0 & 0 \\
\tau_1 & \tau_0 & 0 & \ldots & 0 & 0 \\
\tau_2 & \tau_1 & \tau_0 & \ldots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\tau_{d-1} & \tau_{d-2} & \tau_{d-3} & \ldots & \tau_0 & 0 \\
\tau_d & \tau_{d-1} & \tau_{d-2} & \ldots & \tau_1 & \tau_0 \\
0 & \tau_d & \tau_{d-1} & \ldots & \tau_2 & \tau_1 \\
0 & 0 & \tau_d & \ldots & \tau_3 & \tau_2 \\
0 & 0 & 0 & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \ldots & \tau_d & \tau_{d-1} \\
0 & 0 & 0 & \ldots & 0 & \tau_d
\end{pmatrix}. \tag{17}$$

A multiplier architecture was proposed by E. Mastrovito [11], which reduces this matrix M to a $d \times d$ matrix, Z, using the congruence $X^d \equiv X^{t_{n+i}} + X$. 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 on the left in Figure 4. We remark that the folded matrix, Z (on the right in Figure 4), is very

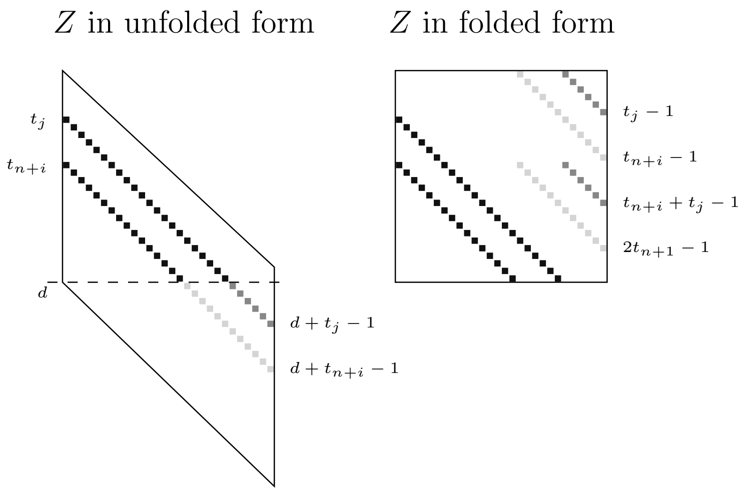

Figure 4: The structures of the unfolded and folded multiplication matrices, for $B_{j,n+i} \bmod T_{n+i}$

sparse. By looking more closely, the congruences

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

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 (16), in Section 4.

4 Analysis of the Algorithms

In order to evaluate precisely the cost of Algorithm 1, we consider equally spaced, consecutive t_i's, 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

$$a = t_1 + (j-1)r, \qquad b = (i-j)r. \tag{18}$$

Note that a randomly chosen set of trinomials having this equally spaced property do not necessarily lead to a valid residue base. We recall that the trinomials, $T_1, \ldots, T_{2n}$ have to be pairwise prime. In Section 5 we give examples of such bases, whose size correspond to extensions of cryptographic interest.

4.1 Complexity analysis for the computation of the ζ_i's

For the first part of the algorithm, i.e., the evaluation of the ζ_i's, 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 (18), the following formulas hold:

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

which, after simplifications, gives

$$\# AND: \quad \frac{r^2 n^2 (n-1)(n+1)}{12}, \tag{19}$$

$$\# XOR: \quad \frac{n(n-1)(r^2n^2 + r^2 n - 2rn - 8r + 18d + 6t_1 + 6)}{12}. \tag{20}$$

For the latency, we remark that the polynomials ζ_i's, 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 $U \times T_{i-1}^{-1} \bmod T_i$, i.e., when $b = r$. (Remember that r is the difference between two consecutive t_i's.) We conclude that the total time complexity for (4) is equal to

$$(n-1)\, T_A + (n-1)\Big(5 + \lceil \log_2((d-1)/r)) \rceil + \lceil \log_2(r) \rceil + \lceil (d-1)/2r \rceil \Big) T_X. \tag{21}$$

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

$t_{n+1} = 2$. [6]

In terms of memory requirements, we have to store polynomials of the form $T_j^{-1}(X) \bmod (X^b+1)$, used to compute (12). 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 $\sum_{i=2}^{n}\sum_{j=1}^{i-1} j\, r = \frac{1}{6}\, rn(n^2-1)$ bits.

4.2 Complexity Analysis for the Computation of the Q_{n+i}'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 $U \times T_j \bmod T_{n+i}$ requires $2d + t_1 + t_{n+1}(i+j-2)r$ AND, and $d + t_1 + t_{n+1} + (i+j-2)r$ XOR.[7] 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 (16) 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),

$$\#\,AND,\ \#\,XOR: \quad \frac{1}{2}n(n-1)(4d + 2rn - 3r + 2t_1 + 2t_{n+1}). \tag{22}$$

The total delay for (16) is thus: $(n-1)(T_A + 3T_X)$.

[6] 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 $2n - 1 < d/2$.

[7] 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_i's in Section 4.1, and for evaluation of the Q_{n+i}'s with Horner's rule in Section 4.2. We have

$$\# AND = \frac{1}{12} n(n-1)(r^2n^2 + 12rn + r^2n + 12t_1 - 18r + 24d + 12t_{n+1}), \tag{23}$$

$$\# XOR = \frac{1}{12} n(n-1)(r^2n^2 + 10rn + r^2n + 18t_1 - 26r + 42d + 12t_{n+1} + 6), \tag{24}$$

with a latency of

$$2(n-1)\,T_A + (n-1)\Big(8 + \lceil \log_2((d-1)/r)\rceil + \lceil \log_2(r)\rceil + \lceil (d-1)/2r\rceil\Big)T_X, \tag{25}$$

or equivalently

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

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 [20], 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 ZU, where Z is a $d \times d$ precomputed matrix[8]; 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 gate count is:

$$\begin{aligned} \# AND: \quad & \frac{1}{6}n\Big(r^2n^3 + 12rn^2 + 12\,(t_1 + t_{n+1} + 2d)\,(n-1) \\ & \qquad - 30rn - r^2n + 30d^2 + 18r\Big), \end{aligned} \tag{26}$$

$$\begin{aligned} \# XOR: \quad & \frac{1}{6}n\Big(r^2n^3 + 10rn^2 + 6n + 6\,(2t_{n+1} + 3t_1)\,(n-1) \\ & \qquad + 42dn - 36rn - r^2n + 30d^2 - 54d - 18 + 26r\Big); \end{aligned} \tag{27}$$

[8] We only need to store $2d$ values per matrix.

and the delay is equal to

$$4n\,T_A + \Big((n-1)\Big(8 + \lceil\log_2(d-1)/r\rceil + \lceil\log_2(r)\rceil + \lceil(d-1)/2r\rceil\Big) + 4\lceil\log_2(d)\rceil + 1\Big)T_X, \quad (28)$$

that we express, for simplicity, as

$$4n\,T_A + \mathcal{O}\left(n + \frac{nd}{r}\right)T_X. \quad (29)$$

5 Discussions and Comparisons

The parameters n, d, and t that appear in the complexity formulae above, make the comparison of our algorithm with previous implementations a difficult task. To simplify, let us assume that $n = k^x$, and $d = k^{1-x}$, (which satisfies $nd = k$). Since we need $2n$ trinomials of degree less than d, having their intermediate coefficient of order less then $d/2$ (see Section 3), the parameters k, x must satisfy $k^{1-2x} > 4$, which, for large values of k, is equivalent to $x < \frac{1}{2}$.[9] 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.[10] For the latency, we remark from Table 1, that the time complexity is mostly influenced by the term in $(d-1)/2b$.

The complexity of Newton's interpolation becomes

$$\#AND: \quad 2k^{1+x} + \frac{r^2}{12}k^{4x} + \mathcal{O}(k^{3x}), \quad (30)$$

and

$$\#XOR: \quad \frac{7}{2}k^{1+x} + \frac{r^2}{12}k^{4x} + \mathcal{O}(k^{3x}); \quad (31)$$

with a latency of

$$2(k^x - 1)\,T_A + \mathcal{O}\left(k^x + k\right)T_X. \quad (32)$$

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

$$\#AND: \quad 5k^{2-x} + 4k^{1+x} + \frac{r^2}{6}k^{4x} + \mathcal{O}(k^{3x}), \quad (33)$$

$$\#AND: \quad 5k^{2-x} + 4k^{1+x} + \frac{r^2}{6}k^{4x} + (2r+2)k^{3x} - \left(\frac{r^2}{6} + 5r + 2\right)k^{2x} + 3rk^x - 4k, \quad (34)$$

and

$$\#XOR: \quad 5k^{2-x} + 7k^{1+x} + \frac{r^2}{6}k^{4x} + \mathcal{O}(k^{3x}), \quad (35)$$

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

[10] We recall that the trinomials have to be relatively prime.

$$\# XOR: \quad 5k^{2-x} + 7k^{1+x} + \frac{r^2}{6}k^{4x} + \left(\frac{5r}{3} + 2\right)k^{3x} - \left(\frac{r^2}{6} + 6r + 1\right)k^{2x} + \left(\frac{13r}{3} - 3\right)k^{x} - 9k \tag{36}$$

for a latency of

$$4k^x\, T_A + \mathcal{O}\left(k\right) T_X. \tag{37}$$

In the literature, the area complexity is usually given according to the number of XOR gates. Most of the studies are dedicated to specific cases, where the reduction polynomial is a trinomial [20], or a pentanomial. E.g., in [17], algorithms for special pentanomials of the form $X^k + X^{t+1} + X^t + X^{t-1} + 1$ are proposed. Our algorithm is a general algorithm, which does not require any special form for the reduction polynomial. The best known general methods have an area complexity of $\mathcal{O}(k^2)$. The best asymptotic area complexity of our algorithm, reached for $x = 2/5$, is in $\mathcal{O}(k^{1.6})$. For completeness, we give the exact complexity formula:

$$\frac{31}{6}k^{8/5} + 7k^{7/5} + \frac{11}{3}k^{6/5} - 9k - \frac{43}{6}k^{4/5} + \frac{4}{3}k^{2/5}. \tag{38}$$

In table 2 below, we count the number of XOR gates for the MMTR algorithm proposed in this paper, and the Montgomery's algorithm proposed in [2]. For all the values of k in Table 2,

$k = n \times d$	MMTR	Montgomery [2]
$168 = 4 \times 42$	38,664	56,616
$180 = 5 \times 36$	37,520	64,980
$192 = 4 \times 48$	49,920	73,920
$234 = 6 \times 39$	54,200	109,746
$252 = 6 \times 42$	62,084	127,260
$360 = 5 \times 72$	139,400	259,560
$486 = 6 \times 81$	213,716	472,878
$567 = 9 \times 63$	212,745	643,345

Table 2: XOR counts for our MMTR and Montgomery's algorithms.

we are able to define a set of $2n$ relatively prime trinomials satisfying $r = 1$, and $t_j < d/2$, for $j = 1, \ldots, 2n$. If we allow r to be greater than 1 in some (very few) cases, then many other interesting decompositions of the extension, k, are possible. We remark that for extensions of cryptographic interest (for ECC), our solution requires fewer XOR gates than Montgomery's algorithm. Note that, in some cases (especially for large values of k), our algorithm also performs better than the pentanomial and even trinomial approaches.

6 Conclusions

We proposed the first general modular multiplication algorithm over finite extension fields, GF(2^k), with subquadratic 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 [17], does not exist.

References

[1] 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.

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

[3] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

[11] 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.

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

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

[14] 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.

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

[16] 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.

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

[18] 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.

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

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

[21] 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.