Subquadratic Polynomial Multiplication over $GF(2^m)$ Using Trinomial Bases and Chinese Remaindering

Éric Schost[1] and Arash Hariri[2]

[1] ORCCA, Computer Science Department, The University of Western Ontario, London, Ontario, Canada
eschost@uwo.ca

[2] Department of Electrical and Computer Engineering, The University of Western Ontario, London, Ontario, Canada
hariri@ieee.org

Abstract. Following the previous work by Bajard-Didier-Kornerup, McLaughlin, Mihailescu and Bajard-Imbert-Jullien, we present an algorithm for modular polynomial multiplication that implements the Montgomery algorithm in a residue basis; here, as in Bajard *et al.*'s work, the moduli are trinomials over $\mathbb{F}_2$. Previous work used a second residue basis to perform the final division. In this paper, we show how to keep the same residue basis, inspired by l'Hospital rule. Additionally, applying a divide-and-conquer approach to the Chinese remaindering, we obtain improved estimates on the number of additions for some useful degree ranges.

Keywords: Montgomery multiplication, Chinese remainder theorem, finite fields, subquadratic area complexity.

1 Introduction

Modular multiplication of polynomials is a cornerstone for many higher-level applications, from finite field arithmetic (for non-prime fields) to implementation of cryptographic protocols. In all that follows, we focus on the practically important case of polynomials over $\mathbb{F}_2$.

Given a polynomial R of degree m, the Montgomery multiplication algorithm [1] shows how to reduce a multiplication modulo a polynomial V of degree at most m to a multiplication modulo R and a division by R, assuming that R and V are coprime. This "multiplication" is slightly twisted, though, since on input A and B, it returns AB/R modulo V.

To implement this algorithm, we need to specify R. An obvious choice is $R = x^m$ [2]. Then, the computations are similar to (but distinct from) those in the Cook-Sieveking-Kung algorithm [3, Chapter 9]. Using fast polynomial multiplication [4,5], this yields an algorithm that uses $O(m \log(m) \log\log(m))$ additions and multiplications. However, the rather large constant hidden in the big-O estimate makes it desirable to devise multiplication schemes with a possibly higher asymptotic cost, but whose performance is better for moderate values of m, say $m \leq 1000$.

To fulfill this goal, we will take another approach to the Montgomery multiplication, that follows the ideas introduced in [6] and [7] (focusing on integer multiplication), and [8] and [9] (for polynomials): we take $R = r_1 \cdots r_n$, with pairwise coprime r_i. To make this approach useful, the computations modulo r_i should be easy: in [9], which focuses on large prime base fields, the r_i's are linear. Here, following [8], the r_i's will be trinomials.

Main Result. Throughout this paper, a family of n pairwise coprime trinomials $\mathscr{R} = (r_1, \ldots, r_n)$ in $\mathbb{F}_2[x]$ is fixed; their product is denoted R. We suppose that all r_i's have the same degree d. This assumption makes it possible for us to give explicit complexity bounds; however, the algorithm still works with trinomials of different degrees. We also assume that all r_i's are squarefree. This can for instance be obtained by taking d odd: in this case, the derivative of $x^d + x^e + 1$ is either x^{d-1} or $x^{d-1} + x^{e-1}$, depending on the parity of e; in both cases, it has no common factor with $x^d + x^e + 1$.

Let $m = nd$ and let V be in $\mathbb{F}_2[x]$, with $\gcd(R, V) = 1$ and $\deg(V) \leq m$; V does not have to be irreducible and can have degree less than m. Computations modulo V will be done through the Montgomery algorithm, applied in the residue basis $\mathscr{R}$; since $m \geq \deg(V)$, a polynomial of degree less than $\deg(V)$ is uniquely determined by its residues modulo $\mathscr{R}$.

Formally, our computational model is the *boolean circuit*, using multiplication (AND) and addition (XOR) gates. The *area complexity* is the number of gates we use; we distinguish between the number of multiplications and additions. The *time complexity* is the length of the longest path in the circuit, i.e., the critical path. As is customary, we write time complexities in the form $\alpha T_A + \beta T_X$, to indicate that all paths in the directed graph underlying the circuit have at most α multiplication gates and β additions gates. Time complexity estimates will depend on a function $T_{\text{rem}}(d)$ defined as follows: $T_{\text{rem}}(d)$ is such that for all $i \leq n$, one can compute the remainder of a polynomial of degree at most $2d - 2$ by r_i using $2d - 2$ additions, in time $T_{\text{rem}}(d)$; we describe this function further in Section 2. Since our main focus is more on the total number of gates than on time complexity, we only give big-O estimates for the latter, except in very simple cases.

Theorem 1. *One can perform modular multiplication in the residue basis* $\mathscr{R} = (r_1, \ldots, r_n)$ *using* $7nd^2$ *multiplications and*

$$7nd^2 + 8n^2d - 2nd\log_2(n) + 6nd - 2n^2 - 10n$$

additions. The time complexity is $O(T_A + \log_2(d)T_X + nT_{\text{rem}}(d))$.

When d is such that one can take $n \simeq d$, our algorithm uses $O(m^{1.5})$ operations. In the worst case, $T_{\text{rem}}(d)$ is in $O(dT_X)$, so the time complexity is $O(T_A + ndT_X)$. If we assume that all trinomials have the form $x^d + x^e + 1$, with $e < d/2$, then $T_{\text{rem}}(d)$ is in $O(T_X)$, and the time complexity is $O(T_A + \log_2(d)T_X + nT_X)$.

Previous Work. There exist several other approaches to modular multiplication; as said above, it is possible to reach a quasi-linear number of operations. Several other families of algorithms are also known, either for low weight moduli [10,11] or for arbitrary ones, such as [12], which shares some features with the family of algorithms we present now.

Our work follows previous results of Bajard *et al.* [8], who use a basis of trinomials as well (with different constraints than ours) and Newton interpolation techniques. Here, we use the Chinese remaindering with a classical divide-and-conquer approach. Mihailescu [9] uses moduli of degree one, whose roots are either roots of unity or consecutive integers; this is not immediately possible here, since we work over $\mathbb{F}_2$.

In both previous papers, a difficulty arises, since the final exact division cannot be performed in the residue basis. The same solutions are used: shifting to another residue basis to do the division. We present an alternative solution, inspired by l'Hospital rule. This enables us to work with the same set of moduli (and thus, to reach higher m for a given moduli degree d), at the cost of a slight increase in the number of operations.

Notation. We write "large" degree polynomials (of degree typically close to $m = nd$) with upper case letters, and "low" degree ones (typically, residues of degree less than d) with lower case letters. Vectors of residues are written in bold face. The equality $A = B \bmod C$ means that A and B are congruent modulo C; the stronger equality $A = B \text{ rem } C$ means that A is the remainder of the division of B by C, so that $\deg(A) < \deg(C)$. The notation $A = B \text{ div } C$ means that A is the quotient in the Euclidean division of B by C.

Outline. Section 2 consists of preliminaries. In Section 3, we consider Chinese remaindering using trinomials. In Section 4, we present our new algorithm and illustrate its performance in Section 5.

2 Preliminaries

This section reviews basic material on operations such as polynomial multiplication or reduction. Most of these results are known; the only new element here is a straightforward estimate on the cost of multiplication by several trinomials.

Polynomial Multiplication. Let a and b be in $\mathbb{F}_2[x]$, of degree less than d. Then, the product ab can be computed using $(d-1)^2$ additions and d^2 multiplications; the time complexity is $T_A + \lceil \log_2(d) \rceil T_X$.

Reduction by Trinomials. For $i \leq n$ and $a \in \mathbb{F}_2[x]$ of degree at most $2d-2$, $a \text{ rem } r_i$ can be computed using $2d-2$ additions [13]. As said before, we write $T_{\text{rem}}(d)$ for the time complexity of this operation; the following estimates for $T_{\text{rem}}(d)$ are available:

– for arbitrary trinomials r_i, we can let $T_{\mathrm{rem}}(d) = (2d-2)T_X$;
– if all trinomials r_i are of the form $x^d + x^{e_i} + 1$, with $e_i < d/2$, then we can take $T_{\mathrm{rem}}(d) = 2T_X$, see [14].

Multiplication by Trinomials. We also need to estimate the cost of multiplication of a polynomial by one or several of the trinomials $r_1, \dots, r_n$. Our result gives a reasonable operation count; however, we are not able to obtain logarithmic time bounds. Such bounds would reduce the overall time complexity of our main algorithm as well.

Proposition 1. *Let P be in $\mathbb{F}_2[x]$ of degree less than s, let $\ell \leq n$ and let $a_1, \dots, a_\ell$ be in $\{1, \dots, n\}$. Then one can compute the product $r_{a_1} \cdots r_{a_\ell} P$ using $2(s-d)\ell + d\ell^2$ additions in time $2\ell T_X$.*

Proof. Let $P_0 = P$ and $P_i = r_{a_i} P_{i-1}$ for $i = 1, \dots, \ell$, so that the polynomial we want to compute is P_n. Remark that P_i has degree less than $s + di$ for all i. Given P_{i-1}, one can compute P_i using $2(s + d(i-1)) - d$ additions. Hence, the total number of additions is at most $2(s-d)\ell + d\ell^2$. Since multiplication by a single trinomial can be done in time $2T_X$, the overall time complexity is $2\ell T_X$. $\square$

3 Chinese Remaindering for Trinomials

We continue with algorithms to perform Chinese remaindering modulo the family of pairwise coprime trinomials $r_1, \dots, r_n$, and for the inverse operation, multiple reduction.

Given residues $\mathbf{a} = (a_1, \dots, a_n)$, with $\deg(a_i) < d$, the Chinese remainder theorem shows that there exists a unique polynomial A of degree less than $m = nd$ with $a_i = A \operatorname{rem} r_i$ for all i. Quasi-linear algorithms of area complexity $O(m \log(m)^2 \log\log(m))$ are known for computing A from its residues a_i, and conversely [3, Chapter 10]. However, the constant hidden in the big-O is rather large (especially for reduction, which uses fast Euclidean division).

These algorithms rely on divide-and-conquer techniques. In what follows, we reuse this idea to devise a Chinese remainder algorithm adapted to trinomials, which performs well for moderate values of m. We also give a (substantially simpler) multiple reduction algorithm with a similar cost.

Linear Combination and Chinese Remaindering. Let $\mathbf{a} = (a_1, \dots, a_n)$ be in $\mathbb{F}_2[x]^n$, with $\deg(a_i) < d$ for all i. We consider here the question of computing the coefficients of the linear combination

$$A = \sum_{i \leq n} a_i S_i, \quad \text{with} \quad S_i = r_1 \cdots r_{i-1} r_{i+1} \cdots r_n;$$

in what follows, we will write $A = \mathrm{LinComb}(\mathbf{a}, \mathscr{R})$. Note that this does not quite solve the Chinese remaindering question, since $A \operatorname{rem} r_i = a_i S_i \operatorname{rem} r_i$: thus, one should divide a_i by S_i modulo r_i prior to the combination. However, in the cases where we apply this algorithm, we will be able to perform this preliminary step jointly with some other operation, so that the main task is indeed the linear combination.

Proposition 2. *Given* $\mathbf{a} = (a_1, \ldots, a_n)$*, one can compute* $\mathrm{LinComb}(\mathbf{a}, \mathscr{R})$ *using* $3n^2d - nd\log_2(n)$ *additions, in time* $O(nT_X)$*.*

Proof. The proof adapts that of [3, Theorem 10.21] to moduli that are trinomials. If $n = 1$, we have nothing to do. Otherwise, let $n' = \lfloor n/2 \rfloor$ and $n'' = n - n'$. Define next

$$B = \sum_{1 \le i \le n'} a_i r_1 \cdots r_{i-1} r_{i+1} \cdots r_{n'}, \quad C = \sum_{n'+1 \le i \le n} a_i r_{n'+1} \cdots r_{i-1} r_{i+1} \cdots r_n,$$

so that we have

$$A = B r_{n'+1} \cdots r_n \ + \ C r_1 \cdots r_{n'}.$$

This leads to a divide-and-conquer algorithm. Assuming that B and C have been computed recursively, A is obtained through multiplications by trinomials, followed by a polynomial addition.

The first step requires to multiply B and C by several trinomials, so it is handled by Proposition 1. Since B has degree less than $n'd$ and we multiply it by n'' trinomials, we obtain a number of additions of $2(n'd - d)n'' + dn''^2$, with a time complexity of $2n''T_X$. Similarly, C has degree less than $n''d$ and we multiply it by n' trinomials, so we get $2(n''d - d)n' + dn'^2$ additions, and a time complexity of $2n'T_X$.

The final polynomial addition takes an extra nd scalar additions, which are done in parallel. After simplifying, we get that the total number of additions needed to reconstruct A from B and C is at most $3n^2d/2 - nd$ for n even, and $3n^2d/2 - nd - d/2$ for n odd. Hence, the number $N(n)$ of additions satisfies the relation:

$$N(n) \le N(n') + N(n'') + 3n^2d/2 - nd.$$

Solving the recurrence gives $N(n) \le 3n^2d - nd\log_2(n)$. Since $2n' \le 2n'' \le n + 1$, the time complexity $D(n)$ satisfies

$$D(n) \le \max(D(n'), D(n'')) + (n + 2)T_X,$$

which yields our claim. □

Reduction. For modular reduction, a more direct approach turns out to work well. In cases where we need the modular reduction in Section 4, the input polynomial A will be even; hence, we present an adapted reduction algorithm, starting with a lemma.

Lemma 1. *Given* A *of degree at most* $s(d-1)$*, one can compute all* $a_i = A \text{ rem } r_i$ *using* $n(s-1)(2d-2)$ *additions, in time* $O(sT_{\mathrm{rem}}(d))$*.*

Proof. We prove that for any given $i \le n$, $a_i = A_i \text{ rem } r$ can be computed using $(s-1)(2d-2)$ additions, in time $(s-1)T_{\mathrm{rem}}(d)$. Doing so in parallel for all r_i proves our proposition.

If $s = 1$, we have nothing to do. Else, we write $A = A_0 + x^{(d-1)(s-2)}A_1$, with $\deg(A_0) < (d-1)(s-2)$ and $\deg(A_1) \le 2(d-1)$. Let $b_i = A_1 \operatorname{rem} r_i$ and $B_i = A_0 + x^{(d-1)(s-2)}b_i$, so that $A = B_i \bmod r_i$ and $\deg(B_i) \le (d-1)(s-1)$. By what was said before, one can compute b_i using $2d-2$ additions, with a time complexity of $T_{\mathrm{rem}}(d)$. Continuing inductively by reducing B_i modulo r_i, the final number of additions is $(s-1)(2d-2)$, and the time is $(s-1)T_{\mathrm{rem}}(d)$. □

Corollary 1. *Let $A \in \mathbb{F}_2[x]$ be even and of degree less than nd. Then, one can compute all $a_i = A \operatorname{rem} r_i$ using $n(n+3)(d-1)$ additions, in time $O(nT_{\mathrm{rem}}(d))$.*

Proof. Let us write $A = B^2$, with $\deg(B) < nd/2$. Our assumptions that all r_i are coprime imply that $n \le d$, so that the latter degree is upper-bounded by $(\lceil n/2 \rceil + 1)(d-1)$. In view of the previous lemma, we can thus compute all $b_i = B_i \operatorname{rem} r_i$ using $n\lceil n/2 \rceil(2d-2) \le 2(n+1)(d-1)$ additions, in time $O(nT_{\mathrm{rem}}(d))$.

Then, we obtain a_i as $b_i^2 \operatorname{rem} r_i$. The cost of reducing b_i^2 modulo r_i is at most $2d-2$, in time $T_{\mathrm{rem}}(d)$. Hence, the total cost is at most $(n+1)(d-1)+2d-2 = (n+3)(d-1)$ additions for reduction by a single trinomial; the time complexity is $O(nT_{\mathrm{rem}}(d))$. □

4 The Multiplication Algorithm

We conclude with presenting our Montgomery-like multiplication algorithm in the residue basis. We start by recalling the Montgomery original construction, then the prior extension to residue basis computations from [9,8], and finally give our new version.

The Montgomery Algorithm. As in the introduction, let R and V be of respective degrees m and m', with $\gcd(R, V) = 1$ and $m' \le m$. Given the inverse W of V modulo R and A, B of degrees less than m', the Montgomery algorithm computes the quantities Z, H, T, Q of Figure 1.

```
Input:
  – $A, B, V, W, R$
Output:
  – $Q = AB/R \bmod V$
1.  $Z = AB$
2.  $H = ZW \operatorname{rem} R$
3.  $T = Z - HV$
4.  $Q = T \operatorname{div} R$.
```

Fig. 1. Montgomery multiplication

Observe that T is 0 modulo R, so that the division yielding Q is exact. Obviously, $Q = AB/R \bmod V$. Besides, since $\deg(R) = m$ and $\deg(T) \le m + m' - 1$, we have $\deg(Q) \le m' - 1$, so $Q = AB/R \operatorname{rem} V$.

The Montgomery Multiplication with Polynomial Residues. In both [9] and [8], the idea of computing modulo a highly composite R is raised. We recall this process here, for our case $R = r_1 \cdots r_n$, with r_i trinomials. Additions and multiplications modulo R are done component-wise modulo $\mathscr{R} = (r_1, \ldots, r_n)$. However, the final step of the algorithm cannot be performed in the residue basis, since it becomes a division by zero.

The workaround in [9,8] consists in shifting from the set of moduli $\mathscr{R}$ to another set $\widetilde{\mathscr{R}} = (r_{n+1}, \ldots, r_{2n})$ modulo which R can be inverted. This shifting process, also called *base extension*, is thus the composite of a Chinese remaindering operation (or Newton interpolation) at $r_1, \ldots, r_n$, followed by a multiple reduction at $r_{n+1}, \ldots, r_{2n}$.

To minimize the overhead, it turns out to be better to take as input the residues $\mathbf{a}, \mathbf{b}$ of A and B modulo $\mathscr{R}$, as well as their residues $\widetilde{\mathbf{a}}, \widetilde{\mathbf{b}}$ modulo $\widetilde{\mathscr{R}}$; similarly, we output the residues $\mathbf{q}, \widetilde{\mathbf{q}}$ of Q modulo both sets. Thus, the algorithm starts as before, performing the computations modulo $\mathscr{R}$. Before the division by R, though, it shifts from the basis $\mathscr{R}$ to $\widetilde{\mathscr{R}}$, divides by R in this basis, and eventually shifts back to $\mathscr{R}$. As input, it also takes the residues $\mathbf{w}$ of W modulo $\mathscr{R}$, and the residues $\widetilde{\mathbf{s}}$ of $S = 1/R$ and $\widetilde{\mathbf{v}}$ of V modulo $\widetilde{\mathscr{R}}$. The details of this algorithm are in Figure 2 (with notation adapted to our setting).

```
Input:
  – $\mathbf{a} = (a_1, \ldots, a_n)$ and $\widetilde{\mathbf{a}} = (a_{n+1}, \ldots, a_{2n})$
  – $\mathbf{b} = (b_1, \ldots, b_n)$ and $\widetilde{\mathbf{b}} = (b_{n+1}, \ldots, b_{2n})$
  – $\mathbf{w} = (w_1, \ldots, w_n)$
  – $\widetilde{\mathbf{s}} = (s_{n+1}, \ldots, s_{2n})$
  – $\widetilde{\mathbf{v}} = (v_{n+1}, \ldots, v_{2n})$
  – $\mathscr{R} = (r_1, \ldots, r_n)$ and $\widetilde{\mathscr{R}} = (r_{n+1}, \ldots, r_{2n})$

Output:
  – $\mathbf{q} = (q_1, \ldots, q_n)$ and $\widetilde{\mathbf{q}} = (q_{n+1}, \ldots, q_{2n})$

1.   $(z_i)_{i \le n}$        $=$  $(a_i b_i \text{ rem } r_i)_{i \le n}$
1'.  $(z_{n+i})_{i \le n}$    $=$  $(a_{n+i} b_{n+i} \text{ rem } r_{n+i})_{i \le n}$
2.   $(h_i)_{i \le n}$        $=$  $(z_i w_i \text{ rem } r_i)_{i \le n}$
2'.  $(h_{n+i})_{i \le n}$    $=$  $\text{shift}(h_1, \ldots, h_n)$
3.   $(t_{n+i})_{i \le n}$    $=$  $(z_{n+i} - h_{n+i} v_{n+i} \text{ rem } r_{n+i})_{i \le n}$
4.   $(q_{n+i})_{i \le n}$    $=$  $(t_{n+i} s_{n+i} \text{ rem } r_{n+i})_{i \le n}$
4'.  $(q_i)_{i \le n}$        $=$  $\text{shift}^{-1}(q_{n+1}, \ldots, q_{2n})$
```

Fig. 2. Residue Montgomery multiplication as in [8,9]

Our Algorithm. Our approach rests on the following remark: when divisions by zero occur, one can still obtain a meaningful result by dividing derivatives. With the notation of Figure 1, from the equality $T = RQ$, we obtain by differentiation

$$T' = R'Q + RQ'.$$

The polynomial R is squarefree, because all r_i are, and are pairwise coprime. Hence, we can deduce the relation

$$Q = \frac{T'}{R'} \bmod R. \tag{1}$$

In contrast to the algorithm of the previous paragraph, our algorithm does not require a second set of moduli: we work with $\mathscr{R} = (r_1, \ldots, r_n)$ all along. Still, as before, we will handle more data as input and output than the mere residues of A and B modulo $\mathscr{R}$. If A is in $\mathbb{F}_2[x]$, we still write its residue representation modulo $\mathscr{R}$ as $\mathbf{a} = (a_1, \ldots, a_n)$. Besides, we denote by $\mathbf{a}^\star$ the residue representation of its derivative, i.e., A':

$$\mathbf{a}^\star = (a_1^\star, \ldots, a_n^\star), \quad \text{with} \quad a_i^\star = A' \bmod r_i.$$

Note that $a_i^\star$ is *not* the derivative of a_i.

The previous algorithm uses a function *shift* to extend the modular information from the moduli $\mathscr{R}$ to $\widetilde{\mathscr{R}}$. In a similar manner, we use a function *diff* that takes as input the residues $\mathbf{a}$ of a polynomial A of degree less than nd, and outputs the residues $\mathbf{a}^\star$ of its derivative: this is done by computing A through the Chinese remaindering, differentiating it, and reducing the result modulo all r_i.

Now, the input of the multiplication algorithm consists of the residues $\mathbf{a}, \mathbf{b}$ of A and B modulo $\mathscr{R}$, and of the residues $\mathbf{a}^\star, \mathbf{b}^\star$ of the derivatives A' and B'; the output consists of the residues $\mathbf{q}$ and $\mathbf{q}^\star$ of Q and its derivative Q'. The computation follows the same steps as before. Since $\mathbf{a}, \mathbf{a}^\star, \mathbf{b}, \mathbf{b}^\star$ are known, we can compute the residues $\mathbf{z}$ and $\mathbf{z}^\star$, using the relations $z_i = a_i b_i$ rem r_i and $z_i^\star = a_i b_i^\star + a_i^\star b_i$ rem r_i.

Next, we deduce the residue representation $\mathbf{h}$ of $H = ZW$ rem R. However, since we take remainders modulo R, the derivative of H cannot be computed term-wise, so we use the function *diff* to obtain $\mathbf{h}^\star$ (which is valid, since $\deg(H) < nd$).

In view of (1), we see that only $\mathbf{t}^\star$ is required to obtain the quotient Q. Since $T = Z - HV$, we deduce that $t_i^\star = T' \bmod r_i$ is given by

$$z_i^\star - h_i v_i^\star - h_i^\star v_i \text{ rem } r_i.$$

Let U be the inverse of R' modulo R and let $\mathbf{u}^\star$ be the residue vector of U modulo $\mathscr{R}$. Equation (1) then implies that $q_i^\star = Q$ rem r_i equals $t_i^\star u_i^\star$ rem r_i. Knowing $\mathbf{q}$, we deduce $\mathbf{q}^\star$ by applying the function *diff* (which is valid, since $\deg(Q) < nd$). Remark that $\mathbf{q}^\star$ is not needed if we perform a single multiplication. However, since Q may be reused for further multiplications, we compute $\mathbf{q}^\star$ for consistency. The details of the algorithm are given in Figure 3.

```
Input:
  – $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{a}^\star = (a_1^\star, \ldots, a_n^\star)$
  – $\mathbf{b} = (b_1, \ldots, b_n)$ and $\mathbf{b}^\star = (b_1^\star, \ldots, b_n^\star)$
  – $\mathbf{w} = (w_1, \ldots, w_n)$
  – $\mathbf{u}^\star = (u_1^\star, \ldots, u_n^\star)$
  – $\mathbf{v} = (v_1, \ldots, v_n)$ and $\mathbf{v}^\star = (v_1^\star, \ldots, v_n^\star)$
  – $\mathscr{R} = (r_1, \ldots, r_n)$

Output:
  – $\mathbf{q} = (q_1, \ldots, q_n)$ and $\mathbf{q}^\star = (q_1^\star, \ldots, q_n^\star)$

1.   $(z_i)_{i \le n}$          =  $(a_i b_i \text{ rem } r_i)_{i \le n}$
1'.  $(z_i^\star)_{i \le n}$    =  $(a_i b_i^\star + a_i^\star b_i \text{ rem } r_i)_{i \le n}$
2.   $(h_i)_{i \le n}$          =  $(z_i w_i \text{ rem } r_i)_{i \le n}$
2'.  $(h_i^\star)_{i \le n}$    =  $\text{diff}(h_1, \ldots, h_n)$
3.   $(t_i^\star)_{i \le n}$    =  $(z_i^\star - h_i v_i^\star - h_i^\star v_i \text{ rem } r_i)_{i \le n}$
4.   $(q_i)_{i \le n}$          =  $(t_i^\star u_i^\star \text{ rem } r_i)_{i \le n}$
4'.  $(q_i^\star)_{i \le n}$    =  $\text{diff}(q_1, \ldots, q_n)$
```

Fig. 3. Our version of the residue Montgomery multiplication

Optimization and Cost Analysis. We finally prove the complexity statement announced in Theorem 1, starting with a discussion of the function *diff*.

This function consists of a Chinese remaindering, followed by differentiation, followed by a multiple reduction. As mentioned in Section 3, the Chinese remaindering requires as a first step the modular multiplication of the residue vector by the vector $(x_i = S_i^{-1} \text{ rem } r_i)_{i \le n}$, with $S_i = r_1 \cdots r_{i-1} r_{i+1} \cdots r_n$. We apply the function diff twice. In both cases, this product can be absorbed in other modular multiplications (requiring us to slightly modify the precomputed polynomials we take as input).

– At step 2', we apply *diff* to the vector $(h_1, \ldots, h_n)$ obtained at step 2. Hence, we can modify step 2, replacing the product $z_i w_i \text{ rem } r_i$ by the product $z_i(w_i x_i \text{ rem } r_i) \text{ rem } r_i$, so that the vector $(w_i x_i \text{ rem } r_i)_{i \le n}$ is needed as input. However, this modifies h_i; since h_i is reused at step 3, we have to compensate for this extra x_i factor: this is done by replacing the product $h_i v_i^\star \text{ rem } r_i$ by $h_i(v_i^\star S_i \text{ rem } r_i) \text{ rem } r_i$, so that we take the latter vector $(v_i^\star S_i \text{ rem } r_i)_{i \le n}$ as input.
– At step 4', we apply *diff* to the vector $(q_1, \ldots, q_n)$ obtained at step 4. Then, we modify step 4, replacing the product $t_i^\star u_i^\star \text{ rem } r_i$ by $t_i^\star(u_i^\star x_i \text{ rem } r_i) \text{ rem } r_i$, so $(u_i^\star x_i \text{ rem } r_i)_{i \le n}$ is used as an extra input.

Hence, the cost of Chinese remaindering reduces to that of the LinComb function given in Proposition 2. Differentiation is free, and gives an even polynomial;

the cost of multiple reduction is given in Corollary 1. Hence, the total cost of *diff* is at most $4n^2d - nd(\log_2(n) - 3) - n^2 - 3n$ additions. The time complexity is in $O(nT_{\mathrm{rem}}(d))$.

We complete the cost analysis of the whole algorithm. The algorithm performs seven vector multiplications in size n, with polynomials of degree less than d: this is done using $7n(d-1)^2$ additions and $7nd^2$ multiplications. There are two calls to *diff*, using $8n^2d - 2nd(\log_2(n) - 3) - 2n^2 - 6n$ additions.

The extra operations are vector additions and remainders in size n. It turns out to be better to postpone the reduction at step 1' to step 3, after all additions are done. Then, we have three size-n additions to perform, on polynomials of degree up to $2d - 2$; hence, they require $3n(2d - 1)$ scalar additions. The four remainders use $4n(2d - 2)$ additions. Summing all previous contributions gives the estimate on the number of operations in Theorem 1.

The time complexity analysis requires no extra complication, except to note that the addition at step 1' can be done in parallel with one at step 3. The total time is then seen to be in $O(T_A + \log_2(d)T_X + nT_{\mathrm{rem}}(d))$.

5 Examples

Table 1 illustrates the number of additions performed by our algorithm for a few values of d. The second column gives the maximal list of trinomials one can use, under the form of a set $S = \{i_1, \ldots, i_n\}$ of integers between 1 and $d - 1$: the corresponding trinomials are $x^d + x^{i_1} + 1, \ldots, x^d + x^{i_n} + 1$. As can be seen, the squarefreeness assumption on our trinomials forces us to discard at least half of the available ones for d even.

Table 1. Numerical examples

d	indices	$n_{\max}$	$n_{\max}\,d$	additions
13	$\{1,2,3,4,6,7,10,12\}$	8	104	15912
14	$\{1,3,5,9,11\}$	5	70	9564
15	$\{1,\ldots,14\} - \{10,12\}$	12	180	35561
16	$\{1,3,5,7,9,13,15\}$	7	112	18691
17	$\{3,4,5,6,9,11,12,14,15\}$	9	153	28919
18	$\{1,3,5,7,9,11,13,15,17\}$	9	162	31768
19	$\{3,4,5,6,7,9,10,12,13,15,16\}$	11	209	45644
20	$\{1,3,5,9,11,15,17\}$	7	140	27325
21	$\{1,\ldots,20\} - \{15\}$	19	399	117393
22	$\{1,3,7,9,11,13,15,19,21\}$	9	198	44428
23	$\{2,3,5,8,9,11,12,14,15,18,20,21,22\}$	13	299	78348
24	$\{5,7,11,13,15,17,19,21,23\}$	9	216	51514
25	$\{1,3,4,7,9,10,13,15,16,18,19,21,22,24\}$	14	350	99352
26	$\{3,5,7,9,11,15,17,21,23\}$	9	234	59104
27	$\{1,\ldots,26\} - \{2,4,9,11,16,18\}$	20	540	186032

Our goal was to obtain a low operation count for multiplication modulo the modulus V. We are successful in this, since our results improve on some of the best ones previously known to us. For instance, Bajard *et al.* [8] have 49920 additions for $m = 192$, 139400 additions for $m = 360$ and 213716 additions for $m = 486$. We obtain 44336 additions for $(n = 8, d = 24, m = nd = 192)$, 108285 additions for $(n = 18, d = 21, m = nd = 378)$ and 159872 additions for $(n = 18, d = 27, m = nd = 486)$.

6 Conclusion

The results given here easily extend to slightly more general situations: e.g., using trinomials of different degrees would enable one to extend and refine the range of accessible degrees. Harder questions concern our time complexity: as of now, our Chinese remaindering or multiple remaindering algorithms have rather bad time complexity, due to their sequential nature. It would be most interesting to obtain a similar operation count with a logarithmic time.

Acknowledgments. We would like to thank the referees for their helpful comments, as well as Paul Zimmermann for pointing out McLaughlin's work. The first author acknowledges the support of the Canada Research Chairs Program and of NSERC.

References

1. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44, 519–521 (1985)
2. Koç, C.K., Acar, T.: Montgomery multiplication in GF(2^k). Designs, Codes and Cryptography 14, 57–69 (1998)
3. von zur Gathen, J., Gerhard, J.: Modern computer algebra. Cambridge University Press, Cambridge (1999)
4. Schönhage, A.: Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. Acta Informatica 7, 395–398 (1977)
5. Cantor, D.G.: On arithmetical algorithms over finite fields. J. Combin. Theory Ser. A 50, 285–300 (1989)
6. Bajard, J.C., Didier, L.S., Kornerup, P.: An RNS Montgomery modular multiplication algorithm. IEEE Transactions on Computers 47, 766–776 (1998)
7. McLaughlin Jr., P.: New frameworks for Montgomery's modular multiplication method. Mathematics of Computation 73, 899–906 (2004)
8. Bajard, J.C., Imbert, L., Jullien, G.A.: Parallel Montgomery multiplication in GF(2^k) using trinomial residue arithmetic. In: 17th IEEE Symposium on Computer Arithmetic, pp. 164–171. IEEE, Los Alamitos (2005)
9. Mihailescu, P.: Fast convolutions meet Montgomery. Mathematics of Computation 77, 1199–1221 (2008)
10. Sunar, B.: A generalized method for constructing subquadratic complexity GF(2^k) multipliers. IEEE Transactions on Computers 53, 1097–1105 (2004)
11. Fan, H., Hasan, M.: A new approach to subquadratic space complexity parallel multipliers for extended binary fields. IEEE Transactions on Computers 56, 224–233 (2007)

12. Giorgi, P., Nègre, C., Plantard, T.: Subquadratic binary field multiplier in double polynomial system. In: SECRYPT 2007 (2007)
13. Wu, H.: Low complexity bit-parallel finite field arithmetic using polynomial basis. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 280–291. Springer, Heidelberg (1999)
14. Ernst, M., Jung, M., Madlener, F., Huss, S., Blümel, R.: A reconfigurable system on chip implementation for elliptic curve cryptography over GF(2^n). In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 381–399. Springer, Heidelberg (2003)