Arithmetic and factorization of polynomials over $\mathbb{F}_2$

JOACHIM VON ZUR GATHEN and JÜRGEN GERHARD*
Fachbereich 17 Mathematik-Informatik
Universität-GH Paderborn
D-33095 Paderborn, Germany
e-mail: {gathen,jngerhar}@uni-paderborn.de

June 14, 1996

Abstract

We describe algorithms for polynomial multiplication and polynomial factorization over the binary field $\mathbb{F}_2$ and their implementation. They allow polynomials of degree up to $100,000$ to be factored in about one day of CPU time, distributing the work on two machines.

1 Introduction

The problem of polynomial factorization over the binary field $\mathbb{F}_2$ is, given a monic polynomial $f \in \mathbb{F}_2[x]$, to compute the factorization $f = f_1^{e_1} \cdots f_r^{e_r}$ with monic irreducible pairwise distinct polynomials $f_1, \ldots, f_r \in \mathbb{F}_2[x]$ and positive $e_1, \ldots, e_r \in \mathbb{N}$. The efficiency of the currently known algorithms for this problem relies on fast polynomial arithmetic, in particular, on fast polynomial multiplication.

The multiplication method of Karatsuba & Ofman (1962) has an asymptotic running time of $O(n^{1.59})$ operations in $\mathbb{F}_2$ for polynomials of degree less than n, which is better than the $O(n^2)$ bound for the naïve multiplication algorithm, but still too slow in practice for large n. Schönhage (1977) gives an $O(n \log n \operatorname{loglog} n)$ algorithm based on a ternary FFT with roots of unity of 3-power order. Reischert (1995) implemented several algorithms for polynomial multiplication over $\mathbb{F}_2$, including those by Karatsuba & Ofman, Schönhage, and Cantor (see below). Shoup (1995) has successfully implemented a fast FFT-based algorithm for multiplying polynomials over $\mathbb{F}_p$ for a prime p, using a modular approach, but it seems to be practical only when p is not too small.

Cantor (1989) presented an algorithm for multiplying polynomials of degree less than n over $\mathbb{F}_p$ for a prime p that uses $O(mp^m)$ multiplications and $O(m^2p^m)$ scalar operations (i.e., additions or multiplications by elements of $\mathbb{F}_p$) in the field $\mathbb{F}_{p^m}$, where m is the least power of p with $4n \le mp^m$. It behaves particularly well in the case $p = 2$. In contrast to the FFT-based algorithms cited above, which evaluate and interpolate at suitably subgroups of the *multiplicative* group of a

*This work is partly supported by the DFG Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen".

(finite) field, Cantor's approach uses *additive* subgroups, i.e., $\mathbb{F}_p$-linear subspaces of $\mathbb{F}_{p^m}$. Montgomery (1991) implemented a polynomial factorization algorithm that uses Cantor's algorithm as a subroutine, and was able to factor a sparse polynomial of degree more than 200,000 over $\mathbb{F}_2$ in about 45 hours.

In section 2, we generalize Cantor's method to work for prime powers p and *arbitrary* m. We state explicit "O"-free upper bounds on the time and space cost of our algorithms. In section 3, simplified versions of the algorithms for $p = 2$ are discussed. In section 4, we present arithmetic circuits for our algorithms. In section 10, we report on experiments in which our implementation of Cantor's original method turned out to be superior to these new variants.

In the last 5 years, dramatic progress in the area of polynomial factorization has been made, both in theory and in practice. The classical algorithms for polynomials over finite fields are due to Berlekamp (1967, 1970), Cantor & Zassenhaus (1981), and Ben-Or (1981). Recently, many variants and asymptotically faster algorithms have been proposed by von zur Gathen & Shoup (1992), Kaltofen (1992), Niederreiter (1994), Gao & von zur Gathen (1994), Kaltofen & Lobo (1994), and Kaltofen & Shoup (1995). Implementations are described in Kaltofen & Lobo (1994), Shoup (1995), and Fleischmann & Roelse (1995).

Section 5 gives an outline of the structure of some modern polynomial factorization algorithms. In sections 6 and 7, we discuss and analyze a new variant of the distinct degree factorization stage in those algorithms, using interval partitions with polynomially growing interval sizes. In sections 8 and 9, we indicate how the distinct degree factorization stage over $\mathbb{F}_2$ can be further speeded up by a special way to compute interval polynomials and by the use of an irreducibility test running in parallel on a second machine. Finally, an implementation of the polynomial factorization algorithm over $\mathbb{F}_2$ is described in section 10, including examples of running times, on pseudorandom polynomials.

We have concentrated on optimizing our implementation for the distinct degree factorization stage. Of course, more work is required to also optimize for cases where the input is known to be special, say when we factor trinomials or cyclotomic polynomials. In particular, we have not yet optimized the equal degree factorization stage of our software.

An Extended Abstract of this paper will appear in Proc. ISSAC '96, ACM Press, Zürich, Switzerland.

2 Fast polynomial multiplication over $\mathbb{F}_q$

Let $\mathbb{F}_q$ be a finite field with q elements and m a positive integer. The extension field $\mathbb{F}_{q^m}$ is an m-dimensional vector space over $\mathbb{F}_q$. Suppose that $W \subseteq \mathbb{F}_{q^m}$ is a fixed k-dimensional subspace, where $1 \leq k \leq m$, and that we want to solve the following problems.

Problem 2.1 *(Multipoint evaluation) Given $f \in \mathbb{F}_{q^m}[x]$ of degree less than q^k, compute $f(\alpha)$ for all $\alpha \in W$.*

Problem 2.2 *(Interpolation) Given a map $\gamma : W \longrightarrow \mathbb{F}_{q^m}$, compute the unique polynomial $f \in \mathbb{F}_{q^m}[x]$ of degree less than q^k satisfying $f(\alpha) = \gamma(\alpha)$ for all $\alpha \in W$.*

The algorithms presented below for these two problems admit a natural parallelization, but here we only discuss the sequential versions.

We fix a basis $(\beta_1, \ldots, \beta_k)$ of W over $\mathbb{F}_q$, and for $0 \leq i \leq k$ let W_i be the subspace $W_i = \operatorname{span}\{\beta_1, \ldots, \beta_i\} \subseteq W$ of dimension i. The sets W_i form a strictly ascending chain

$$\{0\} = W_0 \subsetneq W_1 \subsetneq \cdots \subsetneq W_{k-1} \subsetneq W_k = W, \tag{1}$$

and for $1 \leq i \leq k$ we have the recursive decomposition

$$W_i = \bigcup_{c \in \mathbb{F}_q} (c\beta_i + W_{i-1})$$

of W_i into q pairwise disjoint cosets, which generalizes to

$$\beta + W_i = \bigcup_{c \in \mathbb{F}_q} (\beta + c\beta_i + W_{i-1}) \tag{2}$$

for arbitrary $\beta \in \mathbb{F}_{q^m}$.

Next, we define the sequence of polynomials $s_i \in \mathbb{F}_{q^m}[x]$ for $0 \leq i \leq k$ by

$$s_i = \prod_{\alpha \in W_i} (x - \alpha).$$

Obviously, s_i is a monic squarefree polynomial of degree q^i, and corresponding to (1), we have

$$x = s_0 \mid s_1 \mid \cdots \mid s_{k-1} \mid s_k.$$

The following lemma collects some useful properties of the polynomials s_i.

Lemma 2.3 *The following hold for $0 \leq i \leq k$.*

(i) The recursion formulae

$$s_i = \prod_{c \in \mathbb{F}_q} (s_{i-1} - c s_{i-1}(\beta_i)) = s_{i-1}^q - s_{i-1}(\beta_i)^{q} \cdot s_{i-1}$$

hold if $i \geq 1$.

(ii) s_i is an $\mathbb{F}_q$-linearized polynomial, i.e., $s_i(f+g) = s_i(f) + s_i(g)$ and $s_i(cf) = c s_i(f)$ for all $f, g \in \mathbb{F}_{q^m}[x]$ and $c \in \mathbb{F}_q$.

(iii) $s_i - s_i(\beta) = \prod_{c\in\mathbb{F}_q}(s_{i-1} - s_{i-1}(\beta + c\beta_i)) = \prod_{\alpha\in\beta+W_i}(x-\alpha)$ *for all* $\beta \in \mathbb{F}_{q^m}$.

Note that statement (iii) of the lemma reduces to statement (i) and the definition of the s_i, respectively, if $\beta = 0$.

Proof. We prove i. and ii. by simultaneous induction on i. The case $i = 0$ is immediate from the definitions, so we may assume that $i > 0$. We write $v = s_{i-1}(\beta_i)$ for short. Note that $v \neq 0$ since $\beta_i \notin W_i$.

(i) Using (2) and the linearity of s_{i-1}, we obtain

$$\begin{aligned} s_i &= \prod_{c\in\mathbb{F}_q}\prod_{\alpha\in c\beta_i+W_{i-1}}(x-\alpha) = \prod_{c\in\mathbb{F}_q}\prod_{\alpha\in W_{i-1}}(x - c\beta_i - \alpha) \\ &= \prod_{c\in\mathbb{F}_q}(s_{i-1}(x - c\beta_i)) = \prod_{c\in\mathbb{F}_q}(s_{i-1} - cv) = v^q\prod_{c\in\mathbb{F}_q}(v^{-1}s_{i-1} - c) \\ &= v^q(v^{-q}s_{i-1}^q - v^{-1}s_{i-1}) = s_{i-1}^q - v^{q-1}s_{i-1}, \end{aligned}$$

where we have used Fermat's Little Theorem $x^q - x = \prod_{c\in\mathbb{F}_q}(x-c)$ with $v^{-1}s_{i-1}$ substituted for x in the last line.

(ii) By what we just proved and the induction hypothesis for i., we have

$$\begin{aligned} s_i(f+g) &= s_{i-1}(f+g)^q - vs_{i-1}(f+g) \\ &= \Big(s_{i-1}(f) + s_{i-1}(g)\Big)^q - v\Big(s_{i-1}(f) + s_{i-1}(g)\Big) \\ &= s_{i-1}(f)^q - vs_{i-1}(f) + s_{i-1}(g)^q - vs_{i-1}(g) = s_i(f) + s_i(g), \end{aligned}$$

and

$$\begin{aligned} s_i(cf) &= s_{i-1}(cf)^q - vs_{i-1}(cf) = c^q s_{i-1}(f)^q - vc\cdot s_{i-1}(f) \\ &= c\Big(s_{i-1}(f)^q - vs_{i-1}(f)\Big) = cs_i(f). \end{aligned}$$

(iii) Using i. and ii., we get

$$\begin{aligned} s_i - s_i(\beta) &= s_i(x-\beta) = \prod_{c\in\mathbb{F}_q}(s_{i-1}(x-\beta) - cs_{i-1}(\beta_i)) \\ &= \prod_{c\in\mathbb{F}_q}(s_{i-1} - s_{i-1}(\beta) - cs_{i-1}(\beta_i)) \\ &= \prod_{c\in\mathbb{F}_q}(s_{i-1} - s_{i-1}(\beta + c\beta_i)), \end{aligned}$$

and

$$s_i - s_i(\beta) = s_i(x-\beta) = \prod_{\alpha\in W_i}(x - \beta - \alpha) = \prod_{\alpha\in\beta+W_i}(x-\alpha). \quad \square$$

The decomposition (2) and statement (iii) of the lemma suggest the following algorithm for Problem 2.1. First, we generalize the situation to the evaluation of a polynomial $f \in \mathbb{F}_{q^m}[x]$ of degree less than q^i at all points of the coset $\beta + W_i$, where $0 \le i \le k$ and β is an arbitrary element of W. Problem 2.1 is then the special case where $i = k$ and $\beta = 0$. Next, we partition the coset $\beta + W_i$ into q disjoint cosets $\beta + c\beta_i + W_{i-1}$, one for each $c \in \mathbb{F}_q$, as in (2). For any such c, we divide f with remainder by the polynomial $s_{i-1} - s_{i-1}(\beta + c\beta_i)$, which has exactly the elements of $(\beta + c\beta_i) + W_{i-1}$ as roots, by the previous lemma, and then proceed recursively with the remainder r_c of degree less than q^{i-1} and the coset $(\beta + c\beta_i) + W_{i-1}$. This will yield the desired result since f and r_c have the same values at all points of $(\beta + c\beta_i) + W_{i-1}$.

Algorithm 2.4 *(Multipoint evaluation)*
We assume that the polynomials $s_i \in \mathbb{F}_{q^m}[x]$ for $0 \le i \le k$ as defined above and the values $s_i(\beta_j)$ for $0 \le i < j \le k$ are precomputed and stored.
Input: $i \in \mathbb{N}$ with $0 \le i \le k$, $f \in \mathbb{F}_{q^m}[x]$ of degree less than q^i, and $c_{i+1}, \ldots, c_k \in \mathbb{F}_q$.
Output: The values $f(\alpha)$ for all $\alpha \in \beta + W_i$, where $\beta = \sum_{i<j\le k} c_j\beta_j \in W$.

1. *If $i = 0$ then return f.*
2. *Compute $s_{i-1}(\beta) = \sum_{i<j\le k} c_j s_{i-1}(\beta_j)$, and for each $c \in \mathbb{F}_q$ divide f by $s_{i-1} - s_{i-1}(\beta) - cs_{i-1}(\beta_i)$ with remainder, i.e., compute polynomials $g_c, r_c \in \mathbb{F}_{q^m}[x]$ with*
$$f = g_c(s_{i-1} - s_{i-1}(\beta) - cs_{i-1}(\beta_i)), \ \deg r_c < q^{i-1}.$$
3. *For each $c \in \mathbb{F}_q$, recursively call the algorithm with input $i-1$, r_c and $c, c_{i+1}, \ldots, c_k$ to get $\gamma(\alpha) = r_c(\alpha)$ for all $\alpha \in (\beta + c\beta_i) + W_{i-1}$.*
4. *Return $\gamma(\alpha)$ for $\alpha \in \beta + W_i$.*

Note that $s_{i-1}(\beta) + cs_{i-1}(\beta_i) = s_{i-1}(\beta + c\beta_i)$, by the linearity of s_{i-1}.

Notation. We call an addition in $\mathbb{F}_{q^m}$ or a multiplication of an element of $\mathbb{F}_{q^m}$ by an element of $\mathbb{F}_q$ a *scalar operation*, while a *multiplication* is a multiplication of two arbitrary elements of $\mathbb{F}_{q^m}$. When analyzing the space requirements of an algorithm, we only count the *work space*, i.e., the number of elements of $\mathbb{F}_{q^m}$ for which space is used in addition to the space for input, output, and precomputed objects.

We now give a detailed analysis of the time and space cost of Algorithm 2.4 *without* the precomputation cost, which we will show to be negligible later.

Theorem 2.5 *Algorithm 2.4 works correctly and for $i = k$ uses at most $\frac{q-1}{2}k^2q^k + \frac{q+7}{2}kq^k$ scalar operations and $\frac{q-1}{2}k^2q^k + \frac{q-1}{2}kq^k$ multiplications in $\mathbb{F}_{q^m}$. Furthermore, the algorithm uses work space for at most $\frac{q}{q-1}q^k + 3k$ elements of $\mathbb{F}_{q^m}$.*

Proof. For the correctness proof, we proceed by induction on i. If $i = 0$, then f is a polynomial of degree less than $q^0 = 1$, and hence $f(\alpha) = f$ for all $\alpha \in \mathbb{F}_{q^m}$. So we may assume that $i > 0$. For $\alpha \in (\beta + c\beta_i) + W_{i-1}$ with $c \in \mathbb{F}_q$, we have

$$\begin{aligned}\gamma(\alpha) &= r_c(\alpha) = f(\alpha) - g_c(\alpha)(s_{i-1}(\alpha) - s_{i-1}(\beta) - cs_{i-1}(\beta_i)) \\ &= f(\alpha) - g_c(\alpha)s_{i-1}(\alpha - \beta - c\beta_i) = f(\alpha),\end{aligned}$$

by the induction hypothesis, the linearity of s_{i-1}, and since $\alpha - \beta - c\beta_i \in W_{i-1}$.

For the cost analysis, we note that the polynomials s_i are sparse. It is known that a polynomial $s \in \mathbb{F}_{q^m}[x]$ is linearized with respect to $\mathbb{F}_q$ if and only if the coefficient of x^j in s is zero unless j is a power of q (see, e.g., Lidl & Niederreiter 1983). Hence at most $i + 1$ coefficients of s_i are nonzero. Note that the degree of s_i is q^i, which is exponential in i. The divisor polynomials appearing in step 2 of the algorithm are of the form $s_{i-1} + \tau$ for some constant $\tau \in \mathbb{F}_{q^m}$ and hence have at most i nonzero coefficients besides the leading coefficient, which is one. So one division with remainder of a polynomial of degree less than q^i by $s_{i-1} + \tau$ can be done using no more than $i(q^i - q^{i-1})$ multiplications and $i(q^i - q^{i-1})$ scalar operations in $\mathbb{F}_{q^m}$.

Let $M(i)$ and $S(i)$ denote the number of multiplications and scalar operations in $\mathbb{F}_{q^m}$ the algorithm uses on input i for $0 \le i \le k$, respectively. Then $M(0) = 0 = S(0)$, and for $i > 0$ the cost for the nontrivial steps is:

2. $2(k - i) - 1 + 2(q - 1)$ scalar operations for the computation of $s_{i-1}(\beta)$ and $s_{i-1}(\beta) + cs_{i-1}(\beta_i)$ for $c \in \mathbb{F}_q$, and at most $i(q^i - q^{i-1})$ multiplications and $i(q^i - q^{i-1})$ scalar operations for each of the q divisions with remainder, together no more than
   - $i(q^{i+1} - q^i)$ multiplications and
   - $i(q^{i+1} - q^i) + 2(k - i) + 2q - 3$ scalar operations,
3. $q \cdot M(i - 1)$ multiplications and $q \cdot S(i - 1)$ scalar operations.

Hence we have the recursive inequalities

$$\begin{aligned}M(i) &\le qM(i-1) + i(q^{i+1} - q^i), \\ S(i) &\le qS(i-1) + i(q^{i+1} - q^i) + 2(k - i) + 2q - 3\end{aligned}$$

for $i > 0$. Let $T, U : \mathbb{N} \longrightarrow \mathbb{N}$ be functions with $T(0) = 0$ and $T(i) \le qT(i-1) + U(i)$ for $i > 0$. Unfolding the recursion, we can explicitly bound T in terms of U:

$$T(i) \le \sum_{1\le j\le i} q^{i-j} U(j).$$

This is easily proved by induction on i. Hence

$$M(i) \le \sum_{1\le j\le i} q^{i-j} j(q^{j+1} - q^j) = (q-1)q^i \sum_{1\le j\le i} j = \frac{q-1}{2}(i^2 + i)q^i,$$

and

$$
\begin{aligned}
S(i) &\le \sum_{1\le j\le i} q^{i-j}(j(q^{j+1}-q^j)+2(k-j)+2q-3) \\
&= (q-1)q^i \sum_{1\le j\le i} j + (2(k-i)+2q-3)\sum_{1\le j\le i} q^{i-j} + 2\sum_{1\le j\le i}(i-j)q^{i-j} \\
&= \frac{q-1}{2}(i^2+i)q^i + (2(k-i)+2q-3)\frac{q^i-1}{q-1} + 2\frac{(q-1)iq^i - q(q^i-1)}{(q-1)^2} \\
&= \frac{q-1}{2}i^2q^i + \frac{q-1}{2}iq^i + \frac{2}{q-1}kq^i + \left(\frac{2q-3}{q-1} - \frac{2q}{(q-1)^2}\right)q^i \\
&\quad -\frac{2}{q-1}(k-i) - \left(\frac{2q-3}{q-1} - \frac{2q}{(q-1)^2}\right).
\end{aligned}
$$

Letting $i = k$, we get

$$
\begin{aligned}
M(k) &\le \frac{q-1}{2}k^2q^k + \frac{q-1}{2}kq^k, \\
S(k) &\le \frac{q-1}{2}k^2q^k + \frac{q^2-2q+5}{2(q-1)}kq^k + \frac{2q^2-7q+3}{(q-1)^2}(q^k-1) \\
&\le \frac{q-1}{2}k^2q^k + \frac{q+7}{2}kq^k.
\end{aligned}
$$

Denote by $A(i)$ the work space for elements of $\mathbb{F}_{q^m}$ used by the algorithm on input i. The computation of $s_{i-1}(\beta)$ in step 2 can be done in two storage registers. One division with remainder in step 2 can be carried out in place using one additional register. In a sequential implementation, the storage for the divisions in step 2 and the recursive calls in step 3 can be reused for each $c \in \mathbb{F}_q$, hence space for the output of one division with remainder and three additional registers are sufficient, and we have $A(0) = 0$ and $A(i) \le A(i-1) + q^i + 3$ for $1 \le i \le k$, or

$$
A(k) \le \sum_{1\le j\le k}(q^j+3) = \frac{q^{k+1}-q}{q-1} + 3k \le \frac{q}{q-1}q^k + 3k. \quad \square
$$

The importance of sparse polynomials for fast multipoint evaluation was already remarked by Fiduccia (1972) (in a context where linearized polynomials over finite fields did not occur).

For the interpolation, let $\beta \in W$ and suppose that for each $c \in \mathbb{F}_q$ we have already computed $r_c \in \mathbb{F}_{q^m}[x]$ of degree less than q^{i-1} such that

$$
r_c(\alpha) = \gamma(\alpha) \text{ for all } \alpha \in \beta + c\beta_i + W_{i-1},
$$

or equivalently,

$$
f \equiv r_c \bmod (s_{i-1} - s_{i-1}(\beta) - cs_{i-1}(\beta_i)),
$$

where f is the unique interpolating polynomial of degree less than q^i at all the points of $\beta + W_i$ (note that $s_{i-1}(\beta) + cs_{i-1}(\beta_i) = s_{i-1}(\beta + c\beta_i)$, by the linearity of s_{i-1}). The following lemma gives an explicit formula for f.

Lemma 2.6 *Let F be a field, $n \in \mathbb{N}_{>0}$, $U \subseteq F$ with $\#U = n$, $s \in F[x]$ a nonconstant polynomial, and $(f_u)_{u\in U} \subseteq F[x]$ a sequence of polynomials of smaller degree than s. Then the system of congruences*

$$f \equiv f_u \bmod (s-u) \text{ for all } u \in U \tag{3}$$

has a unique solution $f \in \mathbb{F}[x]$ of degree less than $n \deg s$ which is given by the formula

$$f = \sum_{v\in U} f_v \prod_{\substack{w\in U\\ w\neq v}} \frac{s-w}{v-w}. \tag{4}$$

Note that (4) reduces to the familiar Lagrange interpolation formula when $s = x$.

Proof. First, we show that f as defined in (4) satisfies (3). Fix $u \in U$. Then $\prod_{w\in U,\, w\neq v} \frac{s-w}{v-w}$ is divisible by $s-u$ if $v \neq u$, and hence

$$f \equiv f_u \prod_{\substack{w\in U\\ w\neq u}} \frac{s-w}{u-w} \equiv f_u \prod_{\substack{w\in U\\ w\neq u}} \frac{u-w}{u-w} \equiv f_u \bmod (s-u).$$

The degree of f is clearly less than $n \deg s$. Uniqueness follows from the Chinese Remainder Theorem; we give a proof here for completeness. Suppose that there is another polynomial f' of degree less than $n \deg s$ satisfying $f' \equiv f_u \bmod (s-u)$ for all $u \in U$. Then $f - f'$ is divisible by $s-u$ for all $u \in U$. If $u \neq v$, the two polynomials $s-u$ and $s-v$ are relatively prime since any common divisor divides their difference $u - v \in F \setminus \{0\}$. Therefore, $f - f'$ is divisible by the product polynomial $\prod_{u\in U}(s-u)$ of degree $n \deg s$, and hence is the zero polynomial. □

We write $s = s_{i-1} - s_{i-1}(\beta)$ and $v = s_{i-1}(\beta_i)$ for short. Then, by Lemma 2.6 with $F = \mathbb{F}_{q^m}$, $U = \{cv : c \in \mathbb{F}_q\}$, and $f_{cv} = r_c$, f is given by

$$\begin{aligned} f &= \sum_{c\in\mathbb{F}_q} r_c \prod_{\substack{d\in\mathbb{F}_q\\ d\neq c}} \frac{s-dv}{cv-dv} = \frac{1}{v^{q-1}} \sum_{c\in\mathbb{F}_q} r_c \frac{\prod_{d\in\mathbb{F}_q, d\neq c}(s-dv)}{\prod_{d\in\mathbb{F}_q, d\neq c} c-d} \\ &= \frac{-1}{v^{q-1}} \sum_{c\in\mathbb{F}_q} r_c \frac{\prod_{d\in\mathbb{F}_q}(s-dv)}{s-cv} = \frac{-1}{s_{i-1}(\beta_i)^{q-1}} \sum_{c\in\mathbb{F}_q} r_c \frac{s_i - s_i(\beta)}{s_{i-1} - s_{i-1}(\beta + c\beta_i)}. \end{aligned}$$

In the second line, we used Wilson's identity $\prod_{a\in\mathbb{F}_q^\times} a = -1$ (cf. Lidl & Niederreiter 1983). The last equation follows from Lemma 2.3 (iii). The above discussion leads to the following algorithm.

Algorithm 2.7 *(Interpolation)*
We assume that the polynomials $s_i \in \mathbb{F}_{q^m}[x]$ *for* $0 \le i \le k$ *as defined above and the values* $s_i(\beta_j)$ *and* $-s_i(\beta_{i+1})^{1-q}$ *for* $0 \le i < j \le k$ *are precomputed and stored.*
Input: $i \in \mathbb{N}$ *with* $0 \le i \le k$, $c_{i+1}, \ldots, c_k \in \mathbb{F}_q$, *and an array of values* $\gamma(\alpha) \in \mathbb{F}_{q^m}$, *one for each point* $\alpha \in \beta + W_i$, *where* $\beta = \sum_{i<j\le k} c_j\beta_j \in W$.
Output: The interpolating polynomial $f \in \mathbb{F}_{q^m}[x]$ *of degree less than* q^i *with* $f(\alpha) = \gamma(\alpha)$ *for all* $\alpha \in \beta + W_i$, *and the value* $s_i(\beta)$.

1. *If* $i = 0$ *then return* $\gamma(\beta)$ *and* β.

2. *For each* $c \in \mathbb{F}_q$, *recursively call the algorithm with input* $i-1$, $c, c_{i+1}, \ldots, c_k$, *and* $\gamma(\alpha)$ *for* $\alpha \in (\beta + c\beta_i) + W_{i-1}$ *to get* $r_c \in \mathbb{F}_{q^m}[x]$ *of degree less than* q^{i-1} *with* $r_c(\alpha) = \gamma(\alpha)$ *for all* $\alpha \in (\beta + c\beta_i) + W_{i-1}$, *and* $s_{i-1}(\beta + c\beta_i)$.

3. *Compute* $s_i(\beta) = \sum_{i<j\le k} c_j s_i(\beta_j)$, *and set*

$$f = -s_{i-1}(\beta_i)^{1-q} \sum_{c\in\mathbb{F}_q} r_c \frac{s_i - s_i(\beta)}{s_{i-1} - s_{i-1}(\beta + c\beta_i)}. \tag{5}$$

4. *Return* f *and* $s_i(\beta)$.

Theorem 2.8 *Algorithm 2.7 works correctly and for* $i = k$ *uses no more than* $\frac{q+1}{2}k^2q^k + \frac{q+5}{2}kq^k$ *multiplications and* $\frac{q+1}{2}k^2q^k + \frac{3q+5}{2}kq^k$ *scalar operations in* $\mathbb{F}_{q^m}$. *Furthermore, it uses work space for at most* $\frac{q+1}{q-1}q^k + 3k$ *elements of* $\mathbb{F}_{q^m}$.

Proof. If $i = 0$, then β is the only element of $\beta + W_0 = \beta + \{0\}$, $s_0(\beta) = \beta$, and $\gamma(\beta)$ is the interpolating polynomial of degree less than $q^0 = 1$. The correctness for the case $i > 0$ was already shown in the discussion preceding the algorithm.

Let $S(i)$ and $M(i)$ denote the number of scalar operations and multiplications in $\mathbb{F}_{q^m}$ used by the algorithm on input i, respectively. Then $S(0) = 0 = M(0)$, and for $i > 0$ the cost for the nontrivial steps is:

2. $q \cdot S(i-1)$ scalar operations and $q \cdot M(i-1)$ multiplications,

3. for each $c \in \mathbb{F}_q$ at most

   - $(i+1)q^{i-1}$ multiplications and $(i+1)q^{i-1}$ scalar operations for the multiplication of the polynomial r_c of degree less than q^{i-1} and the sparse polynomial $s_i - s_i(\beta)$,
   - iq^i multiplications and iq^i scalar operations for the subsequent division of the product polynomial of degree less than $q^i + q^{i-1}$ by the sparse polynomial $s_{i-1} - s_{i-1}(\beta + c\beta_i)$ of degree q^{i-1},

   and

- $2(k-i)-1$ scalar operations for the computation of $s_i(\beta)$,
- at most $(q-1)\cdot q^i$ scalar operations for the addition of the q polynomials of degree less than q^i,
- at most q^i multiplications of the coefficients of the result with the precomputed $-s_{i-1}(\beta_i)^{1-q}$,

together no more than

- $i(q^{i+1}+q^i)+2q^i$ multiplications and
- $i(q^{i+1}+q^i)+q^{i+1}+2(k-i)-1$ scalar operations.

This yields the recursive inequalities

$$\begin{aligned} M(i) &\le qM(i-1)+i(q^{i+1}+q^i)+2q^i, \\ S(i) &\le qS(i-1)+i(q^{i+1}+q^i)+q^{i+1}+2(k-i)-1. \end{aligned}$$

The solutions are

$$\begin{aligned} M(i) &\le \sum_{1\le j\le i} q^{i-j}(j(q^{j+1}+q^j)+2q^j) = (q+1)q^i \sum_{1\le j\le i} j + 2iq^i \\ &= \frac{q+1}{2}(i^2+i)q^i+2iq^i, \end{aligned}$$

and

$$\begin{aligned} S(i) &\le \sum_{1\le j\le i} q^{i-j}(j(q^{j+1}+q^j)+q^{j+1}+2(k-j)-1) \\ &= (q+1)q^i\sum_{1\le j\le i} j + iq^{i+1} + (2(k-i)-1)\sum_{1\le j\le i} q^{i-j} + 2\sum_{1\le j\le i}(i-j)q^{i-j} \\ &= \frac{q+1}{2}(i^2+i)q^i + iq^{i+1} + (2(k-i)-1)\frac{q^i-1}{q-1} + 2\frac{(q-1)iq^i - q(q^i-1)}{(q-1)^2} \\ &= \frac{q+1}{2}i^2q^i + \left(\frac{q+1}{2}+q\right)iq^i + \frac{2}{q-1}kq^i - \left(\frac{1}{q-1}+\frac{2q}{(q-1)^2}\right)q^i \\ &\quad -\frac{2}{q-1}(k-i) + \left(\frac{1}{q-1}+\frac{2q}{(q-1)^2}\right). \end{aligned}$$

Finally, for $i=k$ we get

$$\begin{aligned} M(k) &\le \frac{q+1}{2}k^2q^k + \frac{q+5}{2}kq^k, \\ S(k) &\le \frac{q+1}{2}k^2q^k + \frac{3q^2-2q+3}{2(q-1)}kq^k - \frac{3q-1}{(q-1)^2}(q^k-1) \\ &\le \frac{q+1}{2}k^2q^k + \frac{3q+5}{2}kq^k. \end{aligned}$$

Let $A(i)$ denote the work space of the algorithm on input i. The computation of $s_i(\beta)$ in step 3 can be done in two storage registers. In a sequential implementation, the space for the recursive calls of the algorithm in step 2 can be reused for each $c \in \mathbb{F}_q$, and the sum in step 3 can be computed in an accumulating way. The multiplication of one r_c with $s_i - s_i(\beta)$ and the subsequent division with remainder can be carried out in place in the output space of the division, which can also hold the input r_c, using one additional register. The sum is accumulated in the output space for f. Hence space for the output of one division with remainder in step 3 and three additional registers are sufficient, and we have $A(0) = 0$ and $A(i) \le A(i-1) + q^i + q^{i-1} + 3$ for $1 \le i \le k$, or

$$A(k) \le \sum_{1 \le j \le k} (q^j + q^{j-1} + 3) = (q+1)\frac{q^k - 1}{q-1} + 3k \le \frac{q+1}{q-1} q^k + 3k. \quad \square$$

Remark. Cantor (1989) considers the special case where $q = p$ is prime, $k = m = p^d$ is a power of the characteristic, and $W_i = \ker \varphi^i$ for the $\mathbb{F}_p$-endomorphism $\varphi\colon \alpha \mapsto \alpha^p - \alpha$ of $\mathbb{F}_{p^{p^d}}$. He gives a basis $(\beta_1, \ldots, \beta_{p^d})$ of $W = W_k = \mathbb{F}_{p^{p^d}}$ over $\mathbb{F}_p$ such that $W_i = \operatorname{span}\{\beta_1, \cdots, \beta_i\}$ and $\varphi(\beta_i) = \beta_{i-1} \in W_{i-1}$ for $1 \le i \le k$. Then

$$\varphi(s_{i-1}(\beta_i)) = s_{i-1}(\varphi(\beta_i)) = 0,$$

and hence $s_{i-1}(\beta_i) \in \ker \varphi = W_1 = \mathbb{F}_p$, and $s_{i-1}(\beta_i) \ne 0$ since $\beta_i \notin W_{i-1}$. This implies that the polynomials s_i satisfy

$$\begin{aligned} s_0 &= x, \\ s_1 &= x^p - x, \\ s_i &= s_{i-1}^p - s_{i-1}(\beta_i)^{p-1} s_{i-1} = s_{i-1}^p - s_{i-1} = s_1 \circ s_{i-1} \end{aligned}$$

for $i \ge 0$, or more generally $s_i = s_j \circ s_{i-j}$ for $0 \le j \le i$ and $s_i(\alpha) = \varphi^i(\alpha)$ for $\alpha \in \mathbb{F}_{p^m}$. So the coefficients of the s_i are in $\mathbb{F}_p$, which reduces the number of multiplications in $\mathbb{F}_{p^m}$ to $O(kq^k)$. Furthermore, $s_i(\beta_j) = \varphi^i(\beta_j) = \beta_{j-i}$ for $i < j$, and the values $s_i(\beta_j)$ need not even be precomputed.

In the precomputation stage for the algorithms 2.4 and 2.7, the following objects have to be computed.

- The values $s_i(\beta_j)$ for $0 \le i < j \le k$. This can be done by first computing the powers $\beta_j^{q^l}$ for $0 \le l < j$, taking at most $2(j-1)(\lceil \log_2 q \rceil - 1)$ multiplications in $\mathbb{F}_{q^m}$ for β_j, and then multiplying them with the appropriate coefficients of s_i and adding up the results. The total cost for this is at most
  - $$2(\lceil \log_2 q \rceil - 1) \sum_{1 \le j \le k} (j-1) + \sum_{0 \le i < j \le k} i = (\lceil \log_2 q \rceil + 1)(k^2 - k) + \frac{1}{6}(k^3 - k)$$
    multiplications and

– $\sum_{0\le i<j\le k} i = \frac{1}{6}(k^3-k)$ scalar operations in $\mathbb{F}_{q^m}$.

- The values $s_{i-1}(\beta_i)^{q-1}$ for $1\le i\le k$ and the negative of their inverses. This takes at most $2k(\lceil \log_2 q\rceil - 1)$ multiplications, k inversions and k additions in $\mathbb{F}_{q^m}$.

- The polynomials s_i for $0\le i\le k$. By definition, $s_i = s_{i-1}^q - s_{i-1}(\beta_i)^{q-1}s_{i-1}$. Hence the computation of s_i takes at most $2(i-1)(\lceil\log_2 q\rceil - 1)$ multiplications for the qth powers of the $i-1$ nontrivial coefficients of s_{i-1} (recall that s_{i-1} is monic), $i-1$ multiplications of the coefficients of s_{i-1} with $s_{i-1}(\beta_i)^{q-1}$, and $i-1$ additions. Summing over all i, we get at most

  – $(2\lceil\log_2 q\rceil - 1)\sum_{1\le i\le k}(i-1) = (\lceil\log_2 q\rceil - \frac{1}{2})(k^2-k)$ multiplications and

  – $\sum_{1\le i\le k}(i-1) = \frac{1}{2}(k^2-k)$ scalar operations in $\mathbb{F}_{q^m}$.

Hence the total cost for the precomputation stage is at most $\frac{1}{6}k^3 + (2\lceil\log_2 q\rceil - \frac{3}{2})k^2 - \frac{2}{3}k$ multiplications, k inversions, and $\frac{1}{6}k^3 + \frac{1}{2}k^2 + \frac{1}{3}k$ scalar operations in $\mathbb{F}_{q^m}$.

The space occupied by the precomputed objects is i for the nontrivial coefficients of s_i for $1\le i\le k$, one storage location for each $s_i(\beta_j)$ for $0\le i<j\le k$, and one location for $-s_{i-1}(\beta_i)^{1-q}$ for $1\le i\le k$, together

$$\sum_{1\le i\le k} i + \sum_{0\le i<j\le k} 1 + \sum_{1\le i\le k} 1 = 2\frac{k^2+k}{2} + k = k^2+2k.$$

In the following, we consider the problem of polynomial multiplication. Let $a, b\in\mathbb{F}_{q^m}[x]$ be polynomials whose product has degree less than $n\le q^m$. Fix a basis $(b_1,\ldots,b_m)$ of $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$, define the subspaces W_i as above, and precompute the polynomials s_i and the other values needed in the algorithms 2.4 and 2.7 for $k=m$. To compute the product polynomial $c=ab$, first evaluate a and b at all points of W_l, where l is minimal with $n\le q^l$, multiply the $2\cdot q^l$ values pointwise, and interpolate to get c. The cost for this is the cost for two multipoint evaluations and one interpolation at q^l points, plus q^l multiplications in $\mathbb{F}_{q^m}$. We get a total cost of at most

$$\left(2\frac{q-1}{2} + \frac{q+1}{2}\right) l^2q^l + \left(2\frac{q-1}{2} + \frac{q+5}{2}\right) lq^l + q^l \le \frac{3q-1}{2}l^2q^l + \frac{3q+5}{2}lq^l$$

multiplications and

$$\left(2\frac{q-1}{2} + \frac{q+1}{2}\right) l^2q^l + \left(2\frac{q+7}{2} + \frac{3q+5}{2}\right) lq^l = \frac{3q-1}{2}l^2q^l + \frac{5q+19}{2}lq^l$$

scalar operations in $\mathbb{F}_{q^m}$. The work space used in the computation is the output space for the two multipoint evaluations (which can be reused for the output of the pointwise multiplication), plus the maximum of the work spaces for one multipoint evaluation, the pointwise multiplication, and the interpolation, i.e., storage for at most $2q^l + \frac{q+1}{q-1}q^l + 3l = \frac{3q-1}{q-1}q^l + 3l$ elements of $\mathbb{F}_{q^m}$. Using $q^l < qn$, we get the following theorem.

Theorem 2.9 *Two polynomials $a, b \in \mathbb{F}_{q^m}[x]$ whose product has degree less than n, where $q \le n \le q^m$, can be multiplied using less than*

$$\frac{3q^2 - q}{2} n \log_q^2 n + \frac{15q^2 + 7q}{2} n \log_q n$$

multiplications and

$$\frac{3q^2 - q}{2} n \log_q^2 n + \frac{19q^2 + 35q}{2} n \log_q n$$

scalar operations in $\mathbb{F}_{q^m}$, and work space for less than $5qn + 6\log_q n$ elements of $\mathbb{F}_{q^m}$.

If two polynomials a, b over the field $\mathbb{F}_q$ whose product has degree less than n are to be multiplied, we first choose the degree m of the field extension $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$ mimimal so that $n \le \frac{mq^m}{2}$, find an irreducible polynomial $g \in \mathbb{F}_q[x]$ of degree m, and let $\mathbb{F}_{q^m} = \mathbb{F}_q[x]/(g) = \mathbb{F}_q(\alpha)$, where $\alpha = x \bmod g$. Now we regard a, b as "block polynomials", i.e., we write $a = \sum_{0\le j<r} a_j y^j$ with $y = x^{m'}$, $m' = \lceil \frac{m}{2} \rceil$, $a_j \in \mathbb{F}_q[x]$, $\deg a_j < m'$, $(r-1)m' < n \le rm'$, and define $b = \sum_{0\le j<r} b_j y^j$ accordingly. Then

$$c = ab = \sum_{0\le s<r} \underbrace{\left(\sum_{0\le j\le s} a_j b_{s-j} \right)}_{c_s} y^s,$$

with $\deg c_s < 2m' - 1 \le m$. Hence the c_s can be uniquely determined from the coefficients of the product polynomial $\overline{c} = \overline{a} \cdot \overline{b}$ over $\mathbb{F}_{q^m}[y]$ when y is regarded to be an indeterminate, $\overline{a} = \sum_{0\le j<r} a_j(\alpha) y^j$, and $\overline{b}, \overline{c}$ are defined analogously. The degree in y of $\overline{c}$ is at most $r - 1 < \frac{n}{m'} \le \frac{mq^m}{2m'} \le q^m$, and the above method can be applied. After reading off the c_s from the coefficients of $\overline{c}$, they have to be added in an overlapping way to get the coefficients of the original product polynomial c over $\mathbb{F}_q$. The total cost for this is at most

- $\frac{3q-1}{2} m^2 q^m + O(mq^m)$ multiplications in $\mathbb{F}_{q^m}$ for the polynomial multiplication over $\mathbb{F}_{q^m}$,
- $\frac{3q-1}{2} m^2 q^m + O(mq^m)$ scalar operations in $\mathbb{F}_{q^m}$ for the polynomial multiplication, and at most $(r-1)(\lceil \frac{m}{2} \rceil - 1)$ additions in $\mathbb{F}_q$ for the final overlapping addition.

The number of operations in $\mathbb{F}_q$ for a scalar operation in $\mathbb{F}_{q^m}$ is at most m. Let $\mathcal{M}(m)$ denote the number of operations in $\mathbb{F}_q$ used for the multiplication of two polynomials of degree less than m over $\mathbb{F}_q$. Using classical arithmetic, we have $\mathcal{M}(m) = 2m^2 + O(m)$, but we might as well use the method recursively. Then one multiplication in $\mathbb{F}_{q^m}$, which consists of one polynomial multiplication and one polynomial division with remainder over $\mathbb{F}_q$, can be done with $O(\mathcal{M}(m))$ operations in $\mathbb{F}_q$ (cf. Aho *et al.* 1974). Using classical arithmetic, the cost for one multiplication in $\mathbb{F}_{q^m}$ is at most $4m^2 + O(m)$ operations in $\mathbb{F}_q$, and we get a total cost of

$$(6q-2)m^4q^m + O(m^3q^m) \leq (12q^2 - 4q)n\log_q^3 n + O(n\log_q^2 n)$$

operations in $\mathbb{F}_q$, provided that $q < n$. The bound can be improved to $O(n\log_q^2 n(\log_q\log_q n)^3)$ with a recursive approach.

3 Fast polynomial multiplication over $\mathbb{F}_2$

There are several simplifications of the algorithms for $q = 2$, the case we are mainly interested in. In the multipoint evaluation algorithm, it is possible to reduce the number of divisions with remainder from two to one; Cantor (1989) mentions a similar simplification. To see why, let r_0 be the remainder and g be the quotient of the division of f by $s = s_{i-1} + s_{i-1}(\beta_i)$, i.e., $f = gs + r_0$ and $\deg r_0 < 2^{i-1}$. The degrees of the other polynomials are $\deg f < 2^i$, $\deg s = 2^{i-1}$, and $\deg g = \deg f - \deg s < 2^{i-1}$. Now

$$f = g(s + s_{i-1}(\beta_i)) + (r_0 + s_{i-1}(\beta_i)g), \tag{6}$$

and since $r_0 + s_{i-1}(\beta_i)g$ has degree less than 2^{i-1}, it is equal to r_1, the remainder of the division of f by $s + s_{i-1}(\beta_i)$.

In the interpolation algorithm, we have

$$s_i + s_i(\beta) = \prod_{c\in\mathbb{F}_2}(s_{i-1} + s_{i-1}(\beta + c\beta_i)) = (s_{i-1} + s_{i-1}(\beta))(s_{i-1} + s_{i-1}(\beta+\beta_i)),$$

hence no polynomial division is necessary, and the interpolation formula reduces to

$$\begin{aligned} f &= s_{i-1}(\beta_i)^{-1}(r_0(s_{i-1} + s_{i-1}(\beta+\beta_i)) + r_1(s_{i-1} + s_{i-1}(\beta))) \\ &= (s_{i-1} + s_{i-1}(\beta)) \cdot s_{i-1}(\beta_i)^{-1}(r_0 + r_1) + r_0. \end{aligned}$$

Furthermore, both the multipoint evaluation and the interpolation algorithm can be implemented to run in place, i.e., to return the output in the space for the input, using only a small number of work space locations. In the multipoint evaluation algorithm, the division with remainder can be carried out in place, and

r_1 can be computed in the space for the quotient g, using one additional storage register. Similarly, the computation of f from r_0 and r_1 in the interpolation algorithm can be done in the input space containing r_0 and r_1, and one additional register, leaving the result f in the same place.

For completeness, we restate the algorithms for multipoint evaluation and interpolation in the case $q = 2$. As before, we assume that the polynomials s_i for $0 \le i \le k$ and the values $s_i(\beta_j)$ and $s_i(\beta_{i+1})^{-1}$ for $0 \le i < j \le k$ are precomputed.

Algorithm 3.1 *(Multipoint evaluation for $q = 2$)*
Input: $i \in \mathbb{N}$ *with* $0 \le i \le k$, $f \in \mathbb{F}_{2^m}[x]$ *of degree less than* 2^i, *and* $c_{i+1}, \ldots, c_k \in \mathbb{F}_2$.
Output: The values $f(\alpha)$ *for* $\alpha \in \beta + W_i$, *where* $\beta = \sum_{i<j\le k} c_j \beta_j \in W$.

1. *If* $i = 0$ *then return* f.
2. *Compute* $s_{i-1}(\beta) = \sum_{i<j\le k} c_j s_{i-1}(\beta_j)$, *divide* f *by* $s_{i-1} + s_{i-1}(\beta)$ *with remainder, i.e., compute polynomials* $g, r_0 \in \mathbb{F}_{2^m}[x]$ *with*
$$f = g(s_{i-1} + s_{i-1}(\beta)) + r_0 \text{ and } \deg r_0 < 2^{i-1},$$
and set $r_1 = r_0 + s_{i-1}(\beta_i)g$.
3. *For* $c = 0, 1$, *recursively call the algorithm with input* $i - 1$, r_c, *and* $c, c_{i+1}, \ldots, c_k$ *to get* $\gamma(\alpha) = r_c(\alpha)$ *for all* $\alpha \in (\beta + c\beta_i) + W_{i-1}$.
4. *Return* $\gamma(\alpha)$ *for* $\alpha \in \beta + W_i$.

Algorithm 3.2 *(Interpolation for $q = 2$)*
Input: $i \in \mathbb{N}$ *with* $0 \le i \le k$, $c_{i+1}, \ldots, c_k \in \mathbb{F}_2$, *and an array of values* $\gamma(\alpha) \in \mathbb{F}_{2^m}$, *one for each point* $\alpha \in \beta + W_i$, *where* $\beta = \sum_{i<j\le k} c_j \beta_j \in W$.
Output: The interpolating polynomial $f \in \mathbb{F}_{2^m}[x]$ *of degree less than* 2^i *with* $f(\alpha) = \gamma(\alpha)$ *for all* $\alpha \in \beta + W_i$.

1. *If* $i = 0$ *then return* $\gamma(\beta)$.
2. *For* $c = 0, 1$, *recursively call the algorithm with input* $i - 1$, $c, c_{i+1}, \ldots, c_k$, *and* $\gamma(\alpha)$ *for* $\alpha \in (\beta + c\beta_i) + W_{i-1}$ *to get* $r_c \in \mathbb{F}_{2^m}[x]$ *of degree less than* 2^{i-1} *with* $r_c(\alpha) = \gamma(\alpha)$ *for all* $\alpha \in (\beta + c\beta_i) + W_{i-1}$.
3. *Compute* $s_{i-1}(\beta) = \sum_{i<j\le k} c_j s_{i-1}(\beta_j)$, *and set*
$$f = (s_{i-1} + s_{i-1}(\beta)) \cdot s_{i-1}(\beta_i)^{-1}(r_0 + r_1) + r_0.$$
4. *Return* f.

Theorem 3.3 *With the help of the above algorithms, two polynomials $a, b \in \mathbb{F}_{2^m}[x]$ whose product has degree less than $n \leq 2^m$ can be multiplied using less than*

$$\frac{3}{2} n \log_2^2 n + \frac{15}{2} n \log_2 n + 8n$$

multiplications and

$$\frac{3}{2} n \log_2^2 n + \frac{29}{2} n \log_2 n + 4n + 9$$

scalar operations in $\mathbb{F}_{2^m}$. In doing so, work space for less than $2 \log_2 n + 2$ elements of $\mathbb{F}_{2^m}$ is used.

Proof. Denote by $S(i)$ and $M(i)$ the number of scalar operations and multiplications for the multipoint evaluation on input i, respectively. Then we have $S(0) = 0 = M(0)$ and the recursive inequalities

$$\begin{aligned} M(i) &\leq 2M(i-1) + i(2^i - 2^{i-1}) + 2^{i-1} = 2M(i-1) + (i+1)2^{i-1}, \\ S(i) &\leq 2S(i-1) + i(2^i - 2^{i-1}) + 2^{i-1} + (k-i) - 1 \\ &= 2S(i-1) + (i+1)2^{i-1} + (k-i) - 1 \end{aligned} \tag{7}$$

for $i > 0$, which for $i = k$ imply that

$$\begin{aligned} M(k) &\leq \frac{1}{4} k^2 2^k + \frac{3}{4} k 2^k, \\ S(k) &\leq \frac{1}{4} k^2 2^k + \frac{7}{4} k 2^k - 3(2^k - 1). \end{aligned}$$

For the interpolation, we have $S(0) = 0 = M(0)$ and

$$\begin{aligned} M(i) &\leq 2M(i-1) + i2^{i-1} + 2^{i-1} = 2M(i-1) + (i+1)2^{i-1}, \\ S(i) &\leq 2S(i-1) + i2^{i-1} + 2 \cdot 2^{i-1} + (k-i) - 1 \\ &= 2S(i-1) + (i+2)2^{i-1} + (k-i) - 1 \end{aligned}$$

for $i > 0$, and hence

$$\begin{aligned} M(k) &\leq \frac{1}{4} k^2 2^k + \frac{3}{4} k 2^k, \\ S(k) &\leq \frac{1}{4} k^2 2^k + \frac{9}{4} k 2^k - 3(2^k - 1). \end{aligned}$$

The cost for a polynomial multiplication where the product polynomial has degree less than $n \leq 2^m$ is the sum of the costs for the evaluation of the two input polynomials at 2^k points, where k is chosen such that $2^{k-1} < n \leq 2^k$, the pointwise multiplications of the $2 \cdot 2^k$ values, and the final interpolation. Thus we need at most

$$\frac{3}{4} k^2 2^k + \frac{9}{4} k 2^k + 2^k < \frac{3}{2} n \log_2^2 n + \frac{15}{2} n \log_2 n + 8n$$

multiplications and

$$\frac{3}{4}k^2 2^k + \frac{23}{4}k 2^k - 9(2^k - 1) < \frac{3}{2}n \log_2^2 n + \frac{29}{2} n \log_2 n + 4n + 9$$

scalar operations in $\mathbb{F}_{2^m}$.

Let $A(k)$ denote the work space used for two multipoint evaluations, the pointwise multiplication, and one interpolation, where k is as above. The discussion preceding the two algorithms showed that the multipoint evaluation as well as the interpolation steps can be implemented to return the output in the space for the input. In step 2 of Algorithm 3.1 and step 3 of Algorithm 3.2, two storage registers for the computation of $s_{i-1}(\beta)$, and the division with remainder and the multiplication, respectively, are sufficient. Hence both the multipoint evaluation and the interpolation algorithm use work space for $2k$ elements of $\mathbb{F}_{2^m}$. Since $A(k)$ is the maximum of the work space requirements for one multipoint evaluation, the pointwise multiplication, and the interpolation, we have $A(k) = 2k < 2\log_2 n + 2$. $\square$

Remark. In an iterative implementation of the multipoint evaluation algorithm, working "top down" from level k to level 0 of the recursion tree, even 2 registers of work space are sufficient, since they can be reused on any level, and likewise for a "bottom up" iterative implementation of the interpolation algorithm.

4 Arithmetic circuits for multipoint evaluation over $\mathbb{F}_2$

In this section, we present an arithmetic circuit for the multipoint evaluation over $\mathbb{F}_{2^m}$ in the case $m = 4$ graphically. Figures 4.1 and 4.2 show arithmetic circuits for computing the remainders $r_0 = \sum_{0 \le j < 2^i} r_{0,j} x^j$ and $r_1 = \sum_{0 \le j < 2^i} r_{1,j} x^j$ in $\mathbb{F}_{2^m}[x]$ on division of $f = \sum_{0 \le j < 2^{i+1}} f_j x^j \in \mathbb{F}_{2^m}[x]$ by $s_i + a$ and $s_i + a + 1$, respectively, for some $a \in \mathbb{F}_{2^m}$ and Cantor's original polynomials $s_i \in \mathbb{F}_2[x]$ in the cases $i = 3$ and $i = 2, 1, 0$. All operations ($+$ and $\bullet$) are understood in $\mathbb{F}_{2^m}$.

If $g \in \mathbb{F}_{2^m}[x]$ is the quotient of f on division by $s_i + a$, then

$$f = g(s_i + a) + r_0 = g(s_i + a + 1) + (r_0 + g),$$

and hence $r_1 = r_0 + g$. With the exception of $i = 0$, each circuit consist of three stages (shaded in Figures 4.1 and 4.2). The first two stages compute the quotient g and the remainder r_0 on division by $s_i + a$, and the last stage simply adds r_0 to g to get r_1. The second stage is just the first stage shiftet 2^{i-1} lines to the right, and the depth of each of them is equal to the number of nonzero nonleading coefficients of the polynomial $s_i + a$, which is at most $i + 1$.

The first stage performs "one half" of the division: it computes polynomials g' and r' in $\mathbb{F}_{2^m}[x]$ with $f = g'(s_i + a) + r'$, $\deg g' < 2^{i-1}$ and $\deg r' < 3 \cdot 2^{i-1}$. Since the

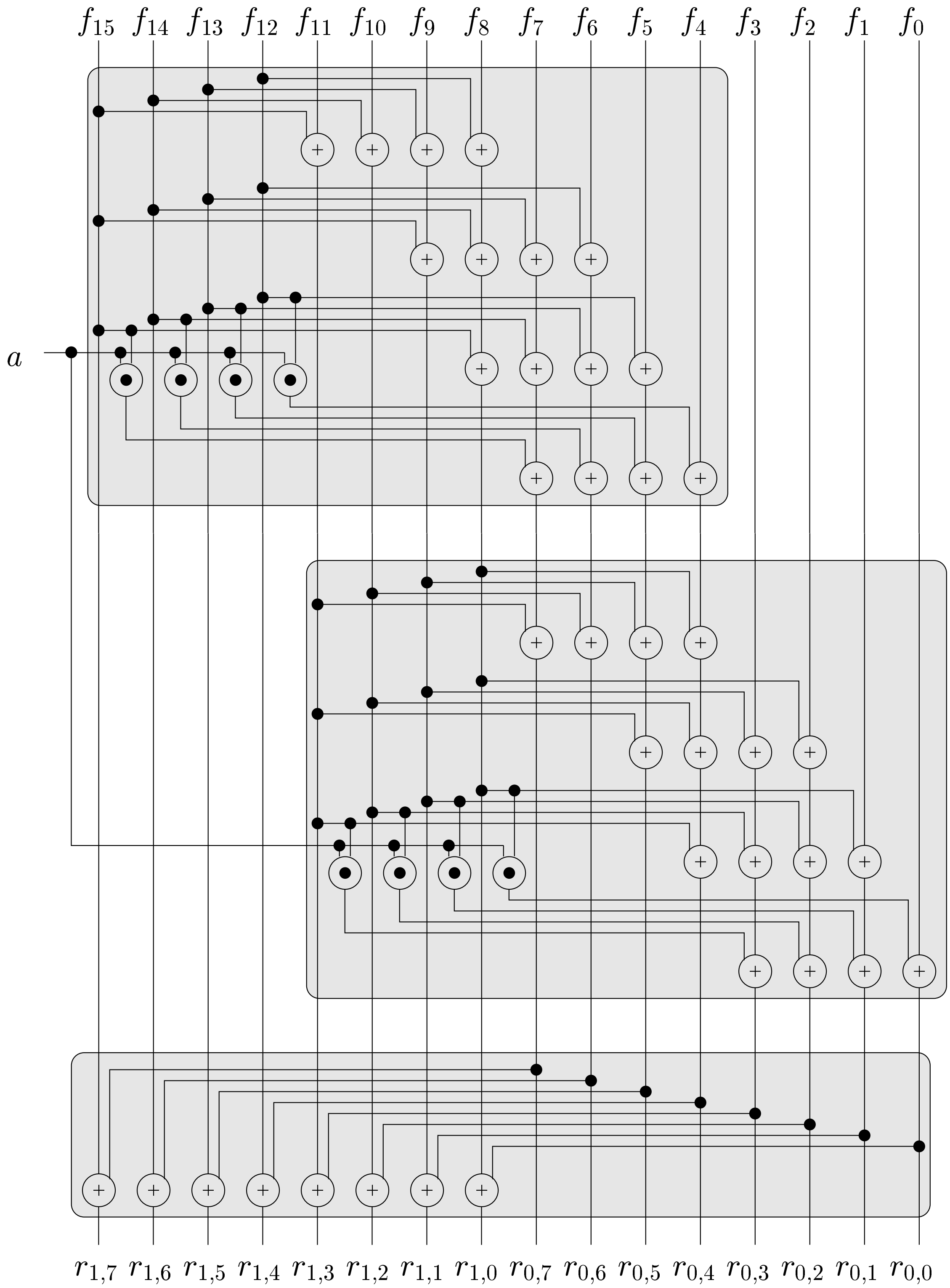

Figure 4.1: Division circuit for $s_3 + a = x^8 + x^4 + x^2 + x + a$

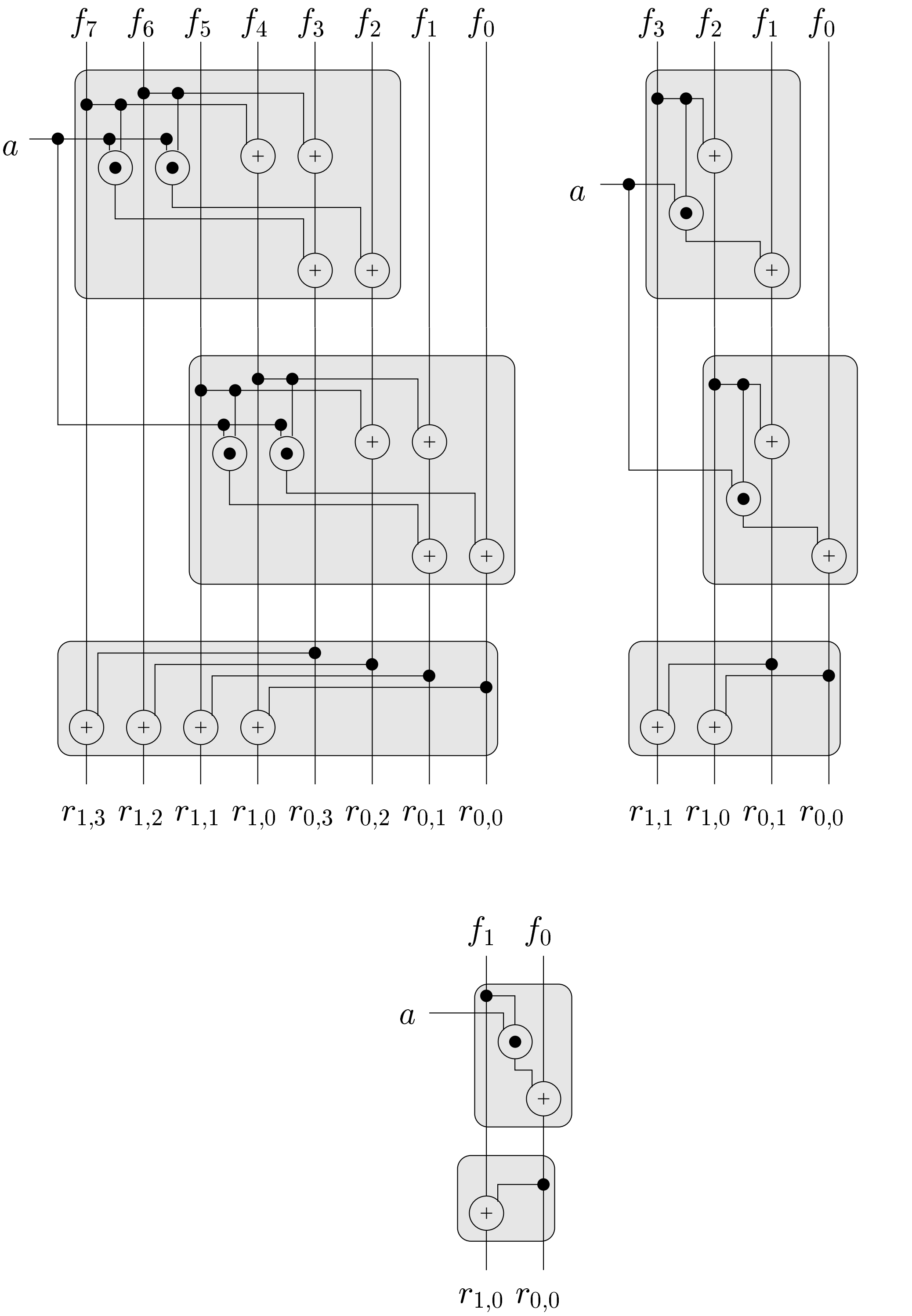

Figure 4.2: Division circuits for $s_2 + a = x^4 + x + a$, $s_1 + a = x^2 + x + a$, and $s_0 + a = x + a$

coefficients of x^j in s_i+a are zero for $2^{i-1} < j < 2^i$, we have $g' = \sum_{0\le j<2^{i-1}} f_{2^i+j}x^j$ and $r' = f + g'(s_i + a)$. The first stage computes the product $g'(s_i + a)$ and adds it to f. There is one level for each nonzero nonleading coefficient of s_i, where no multiplications are necessary since those coefficients are all equal to 1, and the final level is for the constant coefficient a.

The second stage performs the second half of the division in the same way. In the case $i = 0$, there is only one stage for the division at all.

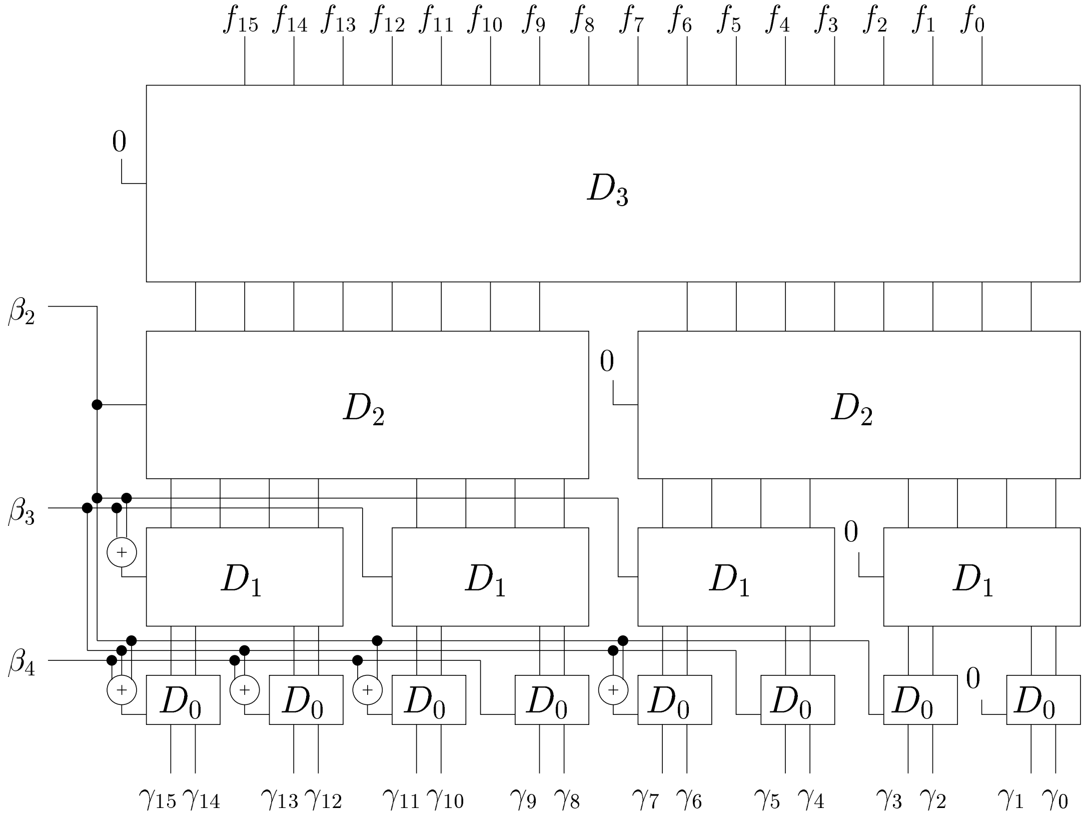

Figure 4.3: Arithmetic circuit for multipoint evaluation over $\mathbb{F}_{16}$

Figure 4.3 shows an arithmetic circuit for evaluating a polynomial $f = \sum_{0\le j<2^4} f_jx^j \in \mathbb{F}_{16}[x]$ at all points of $\mathbb{F}_{16}$, where $\mathbb{F}_{16}$ is represented by an $\mathbb{F}_2$-basis $(\beta_1 = 1, \beta_2, \beta_3, \beta_4) \in \mathbb{F}_{16}^4$ such that the roots of s_i are exactly the elements of span$\{\beta_1, \ldots, \beta_i\}$ for $0 \le i \le 4$. If $\alpha = \sum_{1\le i\le 4} c_i\beta_i \in \mathbb{F}_{16}$, then $f(\alpha) \in \mathbb{F}_{16}$ will be output on line γ_j, where $j = \sum_{1\le i\le 4} c_i 2^{i-1}$ in binary representation.

The structure of the circuit mirrors that of the recursion tree of the multipoint evaluation algorithm. The building blocks are the division circuits from Figures 4.1 and 4.2, where D_{i-1} denotes the circuit for s_{i-1} for $1 \le i \le 4$, plus some additional lines and $+$-gates to compute the constants $a = s_{i-1}(\beta)$ (note that $a + 1 = s_{i-1}(\beta) + s_{i-1}(\beta_i)$ for Cantor's polynomials), which are fed from the left hand side into the division circuits D_{i-1}. If $\beta = c_{i+1}\beta_{i+1} + \cdots + c_m\beta_m$, then

$$\begin{aligned} s_{i-1}(\beta) &= \varphi^{i-1}(\beta) = c_{i+1}\varphi^{i-1}(\beta_{i+1}) + \cdots + c_m\varphi^{i-1}(\beta_m) \\ &= c_{i+1}\beta_2 + \cdots + c_m\beta_{m-i+1}. \end{aligned}$$

It should be noted that the inputs labelled "a" of the division circuits, into which the $s_{i-1}(\beta)$ are fed, do not depend on the particular input polynomial f and hence can be hard-wired into the division circuits themselves. In the cases where $s_{i-1}(\beta) \in \{0, 1\}$, this saves further arithmetic gates. E.g., when $s_{i-1}(\beta) = 0$, the last level of the first and the second stage may be omitted.

The shape of the arithmetic circuits in the general case, i.e., when $m = 2^d$ is an arbitrary power of two, is exactly the same as in the example. The circuit D_i for division with remainder by $s_i + a$ consists of three stages: two for the computation of g and r_0, where the second stage is just the first stage shifted by 2^{i-1} lines to the right, each of them of depth at most $i+1$, and a final addition stage for the computation of r_1. The total depth of D_i is at most $2i+3$.

The circuit for the multipoint evaluation at all the points of $\mathbb{F}_{2^m}$ is built from one D_{m-1}-block, two D_{m-2}-blocks, four D_{m-3}-blocks, etc., and 2^{m-1} D_0-blocks, plus additional lines and $+$-gates for the computation of the $s_{i-1}(\beta)$, along the recursive structure of the algorithm as in Figure 4.3. All arithmetic operations are in $\mathbb{F}_{2^m}$, and the total depth of the circuit is $O(m^2)$ (for Cantor's original algorithm, it is actually $O(3^d)$).

In our algorithm, the nonleading coefficients of the $\mathbb{F}_2$-linearized polynomials s_i no longer need to be 0 and 1 only, but may be arbitrary elements of $\mathbb{F}_{2^m}$. The arithmetic circuits corresponding to the generalized algorithm are very similar to those presented above, with only two differences. First, there are multiplication gates on every level of the first (and also the second) stage of the division circuit D_{i-1}, corresponding to the nonzero nonleading coefficients of s_{i-1}. Second, multiplications by $s_{i-1}(\beta_i)$ have to be added in the third stage to compute r_1 from g and r_0, according to (6). All multiplications are by constants not depending on the particular input polynomial and therefore can be hard-wired into the multipoint evaluation circuit.

The circuits for the interpolation are very similar, they can be obtained by "reversing" the evaluation circuits.

5 Polynomial factorization

Many of the modern polynomial factorization algorithms over finite fields (Cantor & Zassenhaus (1981), Ben-Or (1981), von zur Gathen & Shoup (1992), Kaltofen & Shoup (1995), but not those of Berlekamp (1967, 1970), Gao & von zur Gathen (1994), Kaltofen & Lobo (1994), Niederreiter (1994), and Kaltofen & Shoup (1995), Algorithm B) proceed in three stages:

1. **Squarefree factorization (SFF).** Given a monic polynomial $f \in \mathbb{F}_q[x]$ of degree n, compute the unique monic squarefree and pairwise coprime polynomials $g_1, \ldots, g_n \in \mathbb{F}_q[x]$ such that

$$f = \prod_{1 \leq i \leq n} (g_i)^i.$$

2. **Distinct degree factorization (DDF).** Given a monic squarefree polynomial $f \in \mathbb{F}_q[x]$ of degree n, compute its unique decomposition

$$f = \prod_{1 \leq d \leq n} h_d \tag{8}$$

into monic polynomials $h_1, \ldots, h_n \in \mathbb{F}_q[x]$ such that each h_d has only irreducible factors of degree d. Such an h_d is called an *equal-degree polynomial of order d*.

3. **Equal degree factorization (EDF).** Given integers $d, r \in \mathbb{N}$ with $r \geq 2$ and a squarefree equal-degree polynomial $f \in \mathbb{F}_q[x]$ of order d and degree $n = rd$, compute its r irreducible factors.

In this and the following sections, $\mathsf{M}(n)$ denotes the multiplication time for polynomials over $\mathbb{F}_q$, i.e., two polynomials of degree less than n can be multiplied with $\mathsf{M}(n)$ operations in $\mathbb{F}_q$. Using the approach in section 2, we may e.g. take $\mathsf{M}(n) \in O(n(\log n)^2(\operatorname{loglog} n)^3)$. We also use here that a division with remainder and a gcd for polynomials of degree at most n can be computed using $O(\mathsf{M}(n))$ and $O(\mathsf{M}(n) \log n)$ operations in $\mathbb{F}_q$, respectively (see, e.g., Aho *et al.* 1974, Chapter 8).

Using the deterministic algorithm of Yun (1976), stage 1 can be performed at essentially the cost of one gcd, i.e., $O(\mathsf{M}(n) \log n)$ operations in $\mathbb{F}_q$. The asymptotically fastest of the currently known algorithms for stage 2 when the field size q is fixed is due to Kaltofen & Shoup (1995) and uses $O(n^{1.815} \log q)$ operations in $\mathbb{F}_q$. Finally, stage 3 can be performed with $O(n^{1.7} + \mathsf{M}(n) \log r \log q)$ operations in $\mathbb{F}_q$, using a probabilistic algorithm of von zur Gathen & Shoup (1992).

We now briefly present the algorithm for SFF over the binary field $\mathbb{F}_2$ that we use in our implementation. In fact, it works over any finite field of characteristic two.

Algorithm 5.1 *Squarefree factorization.*
Input: $f \in \mathbb{F}_2[x]$ *monic of degree* $n \in \mathbb{N}$.
Output: $f_1, \ldots, f_n \in \mathbb{F}_q[x]$ *monic, squarefree, and pairwise coprime, such that* $f = \prod_{1 \leq i \leq n} f_i^i$.

1. *Set* $g = \gcd(f, f')$. $\{\ g = (f_2 f_3)^2 (f_4 f_5)^4 (f_6 f_7)^6 \cdots\ \}$

2. *Recursively compute the squarefree factorization* $g_1, \ldots, g_m$ *of* $g^{1/2}$, *where* $2m = \deg g \le n$.
   { $g_i = f_{2i} f_{2i+1}$ *for* $1 \le i \le m$, *where* $f_{n+1} = 1$ *is assumed.* }

3. *Set* $h = f/g$. { $h = f_1 f_3 f_5 f_7 \cdots$ }

4. *Repeat step 5 for* $i = m, \ldots, 1$.

   { $h = \prod_{0 \le j \le i} f_{2j+1}$ }

5. *Set* $f_{2i+1} = \gcd(g_i, h)$, $f_{2i} = g_i / f_{2i+1}$, *and replace* h *by* h/f_{2i+1}.

6. *Set* $f_1 = h$ *and* $f_j = 1$ *for* $2m + 1 < j \le n$.

7. *Return* $f_1, \ldots, f_n$.

The correctness of the algorithm is clear from the comments contained therein, and it can be implemented so as to use $O(\mathsf{M}(n) \log n)$ operations in $\mathbb{F}_2$.

If one factors random polynomials, the dominating cost of the overall algorithm is the cost of the DDF. The reason is that probably an EDF has to be performed only on equal-degree polynomials of small degree (see Flajolet *et al.* (1996) for a detailed analysis). This is confirmed by tests of our factorization routine on random inputs, as reported in section 10.

6 Distinct degree factorization

In the following, we briefly discuss the basic idea of all DDF algorithms over the finite field $\mathbb{F}_q$. We use the following cost measures for our algorithms:

- $\mathsf{P}(n)$, the cost for computing one *modular product* of two polynomials of degree less than n modulo a polynomial of degree at most n. Thus $\mathsf{P}(n) \in O(\mathsf{M}(n))$.

- $\mathsf{Q}(n)$, the cost for one *modular qth power* of a polynomial of degree less than n modulo a polynomial of degree at most n. For large q, we may assume that $\mathsf{Q}(n) \le 2\lfloor \log_2 q \rfloor \mathsf{P}(n)$, while for $q = 2$, modular squaring is cheaper than a general modular multiplication.

- $\mathsf{R}(n)$, the cost for one *remainder computation* of a polynomial of degree at most n modulo a polynomial of degree less than n.

- $\mathsf{D}(n)$, the cost for one *exact division*, i.e., a division where the remainder is known to be zero, of two polynomials of degree at most n.

- $\mathsf{G}(n)$, the cost for one *gcd computation* of two polynomials of degree at most n.

All of the above functions count operations in $\mathbb{F}_q$. This high-level cost analysis is convenient to achieve simultaneously two goals: explicit "O"-free estimates, at least for the dominant term, for various algorithms, and also good asymptotic bounds.

Let $f \in \mathbb{F}_q[x]$ be a monic squarefree polynomial of degree n. It is well known that for any $i \in \mathbb{N}$, $x^{q^i} - x$ is the product of all monic irreducible polynomials in $\mathbb{F}_q[x]$ of degree dividing i (see, e.g., Lidl & Niederreiter 1983, Theorem 3.20). This leads to the following well-known algorithm for computing the DDF (8) of f. For polynomials $a, b \in \mathbb{F}_q[x]$ with $b \neq 0$, we denote the remainder of a modulo b of degree less than $\deg b$ by a rem b.

Algorithm 6.1

Input: A monic squarefree polynomial $f \in \mathbb{F}_q[x]$ of degree n.
Output: The polynomials $h_1, \ldots, h_n \in \mathbb{F}_q[x]$ as in (8).

1. *Set $a_0 = x$, $b_0 = f$, and $m = \lfloor \frac{n}{2} \rfloor$.*
2. *Repeat steps 3 to 5 for $i = 1, \ldots, m$.*
3. $\quad a_i = a_{i-1}^q \operatorname{rem} b_{i-1}$.
4. $\quad h_i = \gcd(a_i - x, b_{i-1})$.
5. $\quad b_i = \dfrac{b_{i-1}}{h_i}$.

   $\{$ *Loop invariants:* $f = b_i \prod_{1 \le l \le i} h_l$ *and* $a_i \equiv x^{q^i} \bmod b_i$. $\}$
6. *Set $h_i = 1$ for $m < i \le n$. If $b_m \neq 1$, set $h_{\deg b_m} = b_m$.*
7. *Return $h_1, \ldots, h_n$.*

The cost of the algorithm is

$$m(\mathsf{Q}(n) + \mathsf{D}(n) + \mathsf{G}(n)) \in O(n \cdot \mathsf{M}(n)(\log q + \log n))$$

operations in $\mathbb{F}_q$. One drawback of the algorithm is that most of the gcds computed will equal 1, since a random polynomial of degree n has about $\log n$ irreducible factors on average. In theory, the cost for a gcd computation for polynomials of degree at most n differs only by a factor of $O(\log n)$ from the cost for a multiplication or a division with remainder, but the constant involved is sufficiently large so that in practice, the computation of a gcd is much more costly than a multiplication or a division with remainder, and much computation time in Algorithm 6.1 is wasted without noticeable progress.

To overcome this problem, some polynomial factorization algorithms (von zur Gathen & Shoup 1992, Kaltofen & Shoup 1995) use a "blocking strategy": the range $\{1, \ldots, \lfloor n/2 \rfloor\}$ for the degrees of possible nontrivial factors of f excepting the one of largest degree is partitioned into intervals $\mathbf{I}_1, \ldots, \mathbf{I}_k$, and there is one gcd computation per interval $\mathbf{I}_j$ which extracts the product of all irreducible factors of f with degree in $\mathbf{I}_j$ (*coarse DDF*). If that gcd turns out to be 1, we know that $h_i = 1$ for all $i \in \mathbf{I}_j$, having computed only one gcd instead of $\#\mathbf{I}_j$ many. If the degree of the gcd is less than $2 \min \mathbf{I}_j$, then we have found an irreducible factor. Otherwise, however, a further step has to be performed to compute the h_i for $i \in \mathbf{I}_j$, e.g., by a linear or binary search of the interval (*fine DDF*).

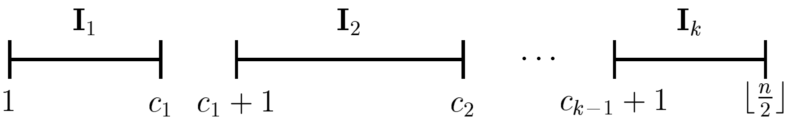

Figure 6.1: An interval partition

We now introduce some notation. Let $m = \lfloor n/2 \rfloor$. An *interval partition* of $\{1, \ldots, m\}$ is a sequence of integers $0 = c_0 < c_1 < c_2 < \cdots < c_{k-1} < c_k = m$, where $0 < k \in \mathbb{N}$ is the *length* of the partition. The sets $\mathbf{I}_j = \{c_{j-1} + 1, \ldots, c_j\}$ for $1 \le j \le k$ are the *intervals* of the partition (see Figure 6.1). For $c, d \in \mathbb{N}$ with $c < d$, we define $(c, d]$ to be the set of polynomials

$$\{f \in \mathbb{F}_q[x] : c < \deg p \le d \text{ for any irreducible factor } p \text{ of } f\}.$$

By an *interval polynomial* for $\mathbf{I}_j$ we mean a polynomial in $(0, c_j]$ which is divisible by each irreducible polynomial in $(c_{j-1}, c_j]$. For example,

$$\prod_{c_{j-1} < i \le c_j} (x^{q^i} - x) \tag{9}$$

is an interval polynomial for $\mathbf{I}_j$, used in von zur Gathen & Shoup (1992). Another example is the polynomial

$$\prod_{0 \le i < c_j - c_{j-1}} (x^{q^{c_j}} - x^{q^i})$$

from Kaltofen & Shoup (1995). Note that interval polynomials need not be squarefree. Kaltofen & Shoup (1995) also use this notion, but their interval polynomials differ from ours in that they are already reduced modulo the polynomial to be factored.

With the above notation, a coarse DDF algorithm can be stated as follows.

Algorithm 6.2 *Coarse DDF.*
Input: A monic squarefree polynomial $f \in \mathbb{F}_q[x]$ of degree n.
Output: The polynomials $H_j = \prod_{i \in \mathbf{I}_j} h_i \in \mathbb{F}_q[x]$ for $1 \leq j \leq k$, where $h_1, \ldots, h_n$ are as in (8), plus an irreducible factor of f of degree more than $\frac{n}{2}$, if such a factor exists.

1. $B_0 = f$.

2. Repeat steps 3 to 5 for $j = 1, \ldots, k$.

3. Compute the remainder I_j of an interval polynomial for $\mathbf{I}_j$ modulo B_{j-1}.

4. $H_j = \gcd(I_j, B_{j-1})$.

5. $B_j = \dfrac{B_{j-1}}{H_j}$.

$$\{ \text{ Loop invariant: } f = B_j \prod_{1 \leq l \leq c_j} h_l. \ \}$$

6. Return $H_1, \ldots, H_k$. If $B_k \neq 1$, then also return B_k.

For constant interval sizes, the above scheme already appears in von zur Gathen & Shoup (1992), Kaltofen & Shoup (1995), and Shoup (1995). Their algorithms only differ in the computation of the interval polynomials. Note that this is exactly the ordinary DDF algorithm when $c_j = j$ for $1 \leq j \leq m$. Intuitively, the interval partition should be chosen in such a way that the intervals increase in size, since a random polynomial has many small but only few large irreducible factors on average.

Using the remainder modulo B_{j-1} of the polynomial (9) for I_j, the cost for the computation of I_j is $(c_j - c_{j-1})$ modular qth powers and the same number of modular multiplications, and hence the cost of the above algorithm is

$$m(\mathsf{P}(n) + \mathsf{Q}(n)) + k(\mathsf{D}(n) + \mathsf{G}(n)) \in O(\mathsf{M}(n)(n \log q + k \log n))$$

operations in $\mathbb{F}_q$. Thus we have reduced the number of gcd computations from m to k in comparison to Algorithm 6.1. The price we pay for this is m additional modular multiplications—which are cheaper than gcd computations—and the fact that we do not yet have the complete DDF of g. If $H_j \neq 1$ and the degree of H_j is less then $2(c_{j-1}+1)$, we know that H_j is irreducible and equal to $h_{\deg H_j}$, and also $h_i = 1$ for $c_{j-1} < i \leq c_j$ with $i \neq \deg H_j$. Otherwise, a fine DDF on H_j will be performed, but the hope is that this will not happen very often, in particular not when H_j has high degree. Of course, whether this is the case heavily depends on the choice of the interval partition.

In practice, the algorithm will be stopped as soon as $\deg B_j < 2(c_{j-1}+1)$, since then B_j must be irreducible and equal to $h_{\deg b_j}$ (this is sometimes called *early abort*), but we do not take this into account in the following analysis in order to keep things simple.

Von zur Gathen & Shoup (1992) and Kaltofen & Shoup (1995) use the interval partition $c_j = lj$ with constant interval sizes and $l = \lceil n^\beta \rceil$ for some real constant β with $0 \le \beta \le 1$. Their algorithms rely on fast multipoint evaluation over the ring $\mathbb{F}_q[x]/(f)$ and on fast matrix multiplication for the computation of the interval polynomials. The cost for the gcd computations in Kaltofen & Shoup (1995) is $O((n^\beta + n^{1-\beta})\mathsf{M}(n)\log n)$.

7 Worst case analysis of the coarse / fine DDF algorithm with polynomially growing interval sizes

In this section, we assume that the interval partition is defined by $c_j = \min\{\lceil j^d \rceil, m\}$ for some $d \in \mathbb{R}$ with $d \ge 1$, where $m = \lfloor n/2 \rfloor$. For simplicity, we assume that $d \in \mathbb{N}$ in the sequel. The number of intervals is $k = \lceil m^{1/d} \rceil$. We will analyze the total cost to compute the complete DDF in the worst case when using a linear search fine DDF algorithm, as follows.

Algorithm 7.1 *Linear search fine DDF.*
Input: $j \in \{1,\ldots,k\}$, *the polynomial* $H_j = \prod_{i \in \mathbf{I}_j} h_i \in \mathbb{F}_q[x]$, *where the* h_i *are as in (8), and* $A_j = x^{q^{c_{j-1}}} \operatorname{rem} H_j \in \mathbb{F}_q[x]$.
Output: The polynomials $h_i \in \mathbb{F}_q[x]$ *for* $i \in \mathbf{I}_j$.

1. *Set* $b_{c_{j-1}} = H_j$ *and* $a_{c_{j-1}} = A_j$.
2. *Repeat steps 3 to 5 for* $i = c_{j-1}+1, \ldots, c_j$.
3. $a_i = a_{i-1}^q \operatorname{rem} b_i$.
4. $h_i = \gcd(a_i - x, b_{i-1})$.
5. $b_i = \dfrac{b_{i-1}}{h_i}$.

   $$\{ \text{ Loop invariants: } H_j = b_i \prod_{c_{j-1} < l \le i} h_l \text{ and } a_i \equiv x^{q^i} \bmod b_i. \ \}$$
6. *Return* $h_{c_{j-1}+1}, \ldots, h_{c_j}$.

This is just the part of the ordinary DDF algorithm for the interval $\mathbf{I}_j$, with the only exception that the polynomial $b_{c_{j-1}}$ at the beginning of the iteration is in $(c_{j-1}, c_j]$ and not only in $(c_{j-1}, n]$. It was also used by von zur Gathen & Shoup (1992) and Kaltofen & Shoup (1995).

An easy calculation shows that

$$\#\mathbf{I}_j = c_j - c_{j-1} \le j^d - (j-1)^d \le dj^{d-1}$$

for $1 \le j \le k$, so that the size of the largest interval in the partition is at most $dk^{d-1} = d\lceil m^{1/d}\rceil^{d-1} \le dn^{(d-1)/d}$ if $n \ge (1-2^{-1/d})^{-d}$. Let $n_j = \deg H_j$ for $1 \le j \le k$. The cost for a fine DDF on the interval $\mathbf{I}_j$ is $(c_j - c_{j-1})(\mathsf{Q}(n_j) + \mathsf{G}(n_j) + \mathsf{D}(n_j))$ operations in $\mathbb{F}_q$. Thus we obtain a total cost of

$$\begin{aligned}
\sum_{1\le j\le k} &(c_j - c_{j-1})(\mathsf{Q}(n_j) + \mathsf{G}(n_j) + \mathsf{D}(n_j)) \\
&\le dn^{(d-1)/d} \sum_{1\le j\le k} (\mathsf{Q}(n_j) + \mathsf{G}(n_j) + \mathsf{D}(n_j)) \\
&\le dn^{(d-1)/d}(\mathsf{Q}(n) + \mathsf{G}(n) + \mathsf{D}(n)) \\
&\in O(n^{(d-1)/d}\mathsf{M}(n)(\log q + \log n))
\end{aligned}$$

operations in $\mathbb{F}_q$ for all intervals, since $\sum_{1\le j\le k} n_j \le n$ and we may assume that $\mathsf{Q}(n)$, $\mathsf{G}(n)$, and $\mathsf{D}(n)$ are superlinear functions of n, i.e., $\mathsf{Q}(n+m) \ge \mathsf{Q}(n)+\mathsf{Q}(m)$ for all sufficiently large $n, m \in \mathbb{N}$, and similarly for G and D.

Since the coarse DDF algorithm already computes the remainder of $x^{q^{c_{j-1}}}$ modulo some multiple of H_j, the cost for the computation of A_j is $\mathsf{R}(n)$ operations in $\mathbb{F}_q$ for one division with remainder by H_j for $1 \le j \le k$. Thus the overall cost for both the coarse and the fine DDF algorithm is at most

$$\frac{n}{2}\mathsf{P}(n) + (\frac{n}{2} + dn^{(d-1)/d})\mathsf{Q}(n) + n^{1/d}\mathsf{R}(n) + (dn^{(d-1)/d} + n^{1/d})(\mathsf{D}(n) + \mathsf{G}(n))$$

operations in $\mathbb{F}_q$ for n large enough. Minimizing the two exponents $(d-1)/d$ and $1/d$ leads to the following result.

Theorem 7.2 *The distinct degree factorization of a squarefree polynomial $f \in \mathbb{F}_q[x]$ of degree n can be computed using $O(n^{1/2}\mathsf{M}(n)\log n)$ operations in $\mathbb{F}_q$ for gcd computations, and $O(n \cdot \mathsf{M}(n)\log q)$ operations in $\mathbb{F}_q$ in total. This can be achieved by means of a coarse DDF algorithm with interval partition defined by $c_j = j^2$ and a fine DDF algorithm with linear interval search.*

Note that this saves a factor of $\log n$ in comparison to the asymptotic running time of Algorithm 6.1, which in particular for $q = 2$ is a significant gain. The same could be achieved with the interval partition $c_j \approx \sqrt{n}j$ as in von zur Gathen & Shoup (1992), but our approach seems to be better suited for random polynomials, which tend to have many irreducible factors of small degree and only few of high degree. The cost for gcd computations in our algorithm is the same as in Kaltofen & Shoup (1995) when $\beta = \frac{1}{2}$, but their total running time is asymptotically smaller due to a faster way of generating the interval polynomials.

It follows from the results in von zur Gathen *et al.* (1995) that for polynomially growing interval sizes, the expected total number of factors in the "bad" intervals is constant for random inputs, so that a fine DDF algorithm with binary interval search promises to save some more costly gcd computations.

8 Computing interval polynomials

In our implementation of the coarse DDF algorithm over $\mathbb{F}_2$, we use the following trick to save multiplications in the computation of interval polynomials. A similar trick has also been used by Montgomery (1991). If $f \in \mathbb{F}_2[x]$ is the polynomial to be factored with no irreducible factors of degree at most c, $n = \deg f$, and we want to compute the remainder modulo f of an interval polynomial for the interval $\{c+1, \ldots, d\} \subseteq \{1, \ldots, \deg f\}$, with c, d divisible by 4 for simplicity, we do the following. Instead of computing the remainder modulo f of the interval polynomial

$$\prod_{c<i\leq d} (x^{2^i} + x) \tag{10}$$

by iterative modular multiplication with $x^{2^i} + x$ for $c < i \leq d$, we compute the remainder modulo f of the polynomial

$$\prod_{c<4i\leq d} H(x^{2^{4i+3}}, x) \tag{11}$$

by iterative modular multiplication with $H(x^{2^{4i+3}}, x)$, where

$$\begin{aligned} H &= \prod_{h\in U} (y-h) = y^8 + H_4 y^4 + H_2 y^2 + H_1 y + H_0 \in \mathbb{F}_2[x,y], \\ U &= \{x, x^2, x^4, x^8, x+x^2+x^4, x+x^2+x^8, x+x^4+x^8, x^2+x^4+x^8\}, \end{aligned}$$

and $H_0, H_1, H_2, H_4 \in \mathbb{F}_2[x]$ are of degree at most 43. The polynomial H is an $\mathbb{F}_2$-linearized polynomial over $\mathbb{F}_2[x]$, and U is a coset of a three-dimensional $\mathbb{F}_2$-subspace of $\mathbb{F}_2[x]$. Since the multiplications by the "small" polynomials H_0, H_2, H_2, H_4 essentially cost the same as a scalar operation, the dominant cost in the computation of $H(x^{2^{4i+3}}, x)$ rem f is $4Q(n)$. Thus the computation of the polynomial (11) modulo f costs essentially $(\frac{d-c}{4} - 1)\mathsf{P}(n) + (d-c)\mathsf{Q}(n)$ operations in $\mathbb{F}_2$, in contrast to $(d-c-1)\mathsf{P}(n) + (d-c)\mathsf{Q}(n)$ operations for the polynomial (10).

We note that the polynomial (11) is not an interval polynomial in general. The factors

$$(y+x+x^2+x^4)(y+x+x^2+x^8)(y+x+x^4+x^8)(y+x^2+x^4+x^8)$$

of H, whose sole purpose is to make H a linearized polynomial (if H were not linearized, this would imply additional modular multiplications), may give raise to irreducible "phantom" factors in the gcd of f and (11) of degree outside the interval $\{c+1, \ldots, d\}$. Since f has no irreducible factors of degree at most c, it is only possible that phantom factors of degree more than d occur. The phantom factors are detected in the fine DDF algorithm, their product is singled out and factored recursively. We guess that phantom factors rarely occur for random polynomials, and in fact they have never occurred in any of our experiments.

In the fine DDF algorithm, we compute the remainder modulo f of the interval polynomial

$$\prod_{c<2i\leq d} (x^{2^{2i}} + x)(x^{2^{2i}} + x^2) \bmod f = \prod_{c\leq i<d} (x^{2^i} + x) \cdot \prod_{c<2i\leq d} (x^{2^{2i-1}} + x)$$

by iterative modular multiplication with $(x^{2^{2i}} + x)(x^{2^{2i}} + x^2) = x^{2^{2i+1}} + (x^2 + x)x^{2^{2i}} + x^3$, at a cost of essentially $(\frac{d-c}{2} - 1)\mathsf{P}(n) + (d-c)\mathsf{Q}(n)$ operations in $\mathbb{F}_2$. Here, no phantom polynomials like in the coarse DDF algorithm can occur.

9 Employing irreducibility tests

In this section, we indicate how to speed up the coarse DDF algorithm by using an irreducibility test. Suppose that we have already found all irreducible factors of $f \in \mathbb{F}_2[x]$ of degree at most $c \in \mathbb{N}$, and that a "large" factor $b \in \mathbb{F}_2[x]$ collecting all irreducible factors of f of degree more than c remains. If b is irreducible, then the coarse DDF algorithm will only find this out after reaching the degree $\deg b/2$, and this may take most of the total time spent for factoring f, as our experiments in section 10 indicate.

In our implementation, we run an irreducibility test on b in parallel to the coarse DDF algorithm on a second machine. If the coarse DDF algorithm finds a factor, the irreducibility test is aborted and restartet for the remaining polynomial. If, however, the irreducibility test says that the polynomial is irreducible, then the coarse DDF algorithm is aborted. The irreducibility test we use is based on Fact 7.3 and Theorem 7.5 in von zur Gathen & Shoup (1992), has an asymptotic running time of $O((n^2 + n^{1/2} \cdot \mathsf{M}(n)) \log^2 n / \operatorname{loglog} n)$ operations in $\mathbb{F}_2$ for a squarefree polynomial in $\mathbb{F}_2[x]$ of degree n, and uses space for $O(n^{3/2})$ elements of $\mathbb{F}_2$, where $n = \deg b$. It involves the computation of matrix products of size $n/2 \times n/2$, which we compute with $O(n^{3/2})$ operations in $\mathbb{F}_2$, using classical matrix arithmetic. The same technique was also used in Shoup (1995) for the computation of modular compositions. Asymptotically, the total time for the irreducibility test is almost the same order of magnitude as the $O(n \cdot \mathsf{M}(n))$ bound for the coarse DDF algorithm, but in our implementation it significantly reduces the total running time. We have further speeded up the irreducibility test by keeping large amounts (growing quadratically with the input degree) of intermediate data in secondary storage, as follows.

In the course of the coarse DDF algorithm, the remainders of the polynomials $x^2, x^4, x^8, \ldots, x^{2^c}$ modulo multiples of b are computed and written to disk. The irreducibility test from von zur Gathen & Shoup (1992), applied to b, computes $x^{2^{\deg b}} \bmod b$ and $x^{2^{\deg b/t}} \bmod b$ for all prime divisors t of $\deg b$ with $\deg b/t > c$. To compute $x^{2^e} \bmod b$ for some $e \in \mathbb{N}$, we use the following variant of von zur Gathen & Shoup's algorithm.

Algorithm 9.1 *Frobenius powers.*
Input: $b \in \mathbb{F}_2[x]$ *and* $e \in \mathbb{N} \setminus \{0\}$. *We assume that the remainders of the polynomials* $x^2, x^4, x^8, \ldots, x^{2^c}$ *modulo multiples of* b *have already been computed for some* $c \in \mathbb{N}$.
Output: $g \in \mathbb{F}_2[x]$ *with* $\deg g < \deg b$ *and* $g \equiv x^{2^e} \bmod b$.

1. *Write* $e = \sum_{0 \le j \le l} e_j 2^{l-j}$ *with* $e_j \in \{0,1\}$ *for* $0 \le j \le l$ *and* $e_0 = 1$.
2. *Let* $k \in \{0, \ldots, l\}$ *be minimal such that* $\lfloor e/2^k \rfloor = \sum_{0 \le j \le l-k} e_j 2^{l-k-j} \le c$, *and set* $g_{l-k} = x^{2^{\lfloor e/2^k \rfloor}} \operatorname{rem} b$, *performing one table-lookup and one subsequent division with remainder by* b.
3. *Repeat step 4 for* $i = l-k+1, \ldots, l$.
4. *Set* $h_i = g_{i-1}(g_{i-1}) \operatorname{rem} b$, *using one modular composition.*
5. *If* $e_i = 0$ *then set* $g_i = h_i$ *else set* $g_i = h_i^2 \operatorname{rem} b$.

   $\{ \; g_i \equiv x^{2^{\lfloor e/2^{l-i} \rfloor}} \bmod b \; \}$
6. *Return* g_l.

The cost of the above algorithm is essentially at most $\lfloor \log_2 \frac{e}{c+1} \rfloor + 1$ modular compositions, since the cost for the initial division with remainder and the modular squarings are negligible. So if c is close to $\deg b$, only very few modular compositions are sufficient to test b for irreducibility, in particular when $\deg b$ has no small prime factors.

Since the space occupied by the the data written to disk grows quadratically with the input size, the practicability of the above method is limited by the amount of disk space available. For degrees above 131072, the table size easily exceeds 1024M bytes. One can still, however, store as much as possible in that case.

We note that the irreducibility test can not only check whether b is irreducible or not but can also detect if b is an equal-degree polynomial of order dividing $\deg b$, using Fact 7.4 in von zur Gathen & Shoup (1992).

10 Implementation and running times

In this section, we describe our implementation of the polynomial factorization algorithm over $\mathbb{F}_2$ on two Sun Sparc Ultra 1 computers, rated at 143 MHz each. The software is written in `C++`. Polynomials over $\mathbb{F}_2$ are represented as arrays of 32-bit unsigned integers, and 32 consecutive coefficients of a polynomial are packed into one machine word. We built a `C++` class for polynomials over $\mathbb{F}_2$ offering standard operations like copying, reversing, shifting, and determining the degree of polynomials, the arithmetic operations addition, multiplication, squaring, division with remainder, and the Extended Euclidean Algorithm.

Polynomial multiplication. We have implemented several algorithms for polynomial multiplication over $\mathbb{F}_2$: the school method, Karatsuba & Ofman, Cantor's method over $\mathbb{F}_{2^{16}}$ and over $\mathbb{F}_{2^{32}}$, and our extension of Cantor's method over $\mathbb{F}_{2^{20}}$. We did not implement Schönhage's algorithm. The timings of Reischert (1995) indicate that in his implementation, it beats Cantor's method for degrees above $500,000$, and for degrees around $40,000,000$, Schönhage's algorithm is faster than Cantor's by a factor of $\approx \frac{3}{2}$.

As basis for the classical multiplication and the method of Karatsuba & Ofman, we have tried several algorithms for the multiplication of polynomials of degree less than 32: by direct classical multiplication, by 3 classical multiplications of 16-bit blocks à la Karatsuba & Ofman, and by 9 multiplications of 8-bit blocks à la Karatsuba & Ofman, where the 8-bit blocks are multiplied via table-lookup (the corresponding table uses 128k bytes of main memory). The last variant has turned out to be the fastest.

Multiplication in $\mathbb{F}_{2^{16}}$ is implemented by means of an exponentiation and a discrete logarithm table with respect to a primitive element of the multiplicative group. This was also done by Montgomery (1991) and Reischert (1995). The cost for one multiplication in $\mathbb{F}_{2^{16}}$ is then essentially the cost for three table lookups and one addition of 16-bit integers. The size of each of the two tables is 256k bytes of main memory. Multiplication in $\mathbb{F}_{2^{20}}$ is done in the same way, using an exponentiation and a logarithm table of size 4M bytes each. For multiplications in $\mathbb{F}_{2^{32}}$, the corresponding tables are far too large to fit in main memory, and we use an alternate approach.

We use two representations for elements of $\mathbb{F}_{2^{32}}$. The first is the usual polynomial basis representation $\sum_{0\le i<32} a_i\alpha^i$, where $\alpha = x \bmod G$ for an irreducible generating polynomial $G \in \mathbb{F}_2[x]$ of degree 32 for $\mathbb{F}_{2^{32}}$. This will be referred to as *representation A* in the sequel. The second is a polynomial basis representation of $\mathbb{F}_{2^{32}}$ over $\mathbb{F}_{2^{16}}$. We have chosen the primitive polynomial $g = x^{16} + x^{15} + x^{13} + x^4 + 1 \in \mathbb{F}_2[x]$ as generating polynomial of $\mathbb{F}_{2^{16}}$ over $\mathbb{F}_2$, and write elements of $\mathbb{F}_{2^{16}}$ in the polynomial basis representation $\sum_{0\le i<16} b_i\beta^i$ with $\beta = x \bmod g \in \mathbb{F}_{2^{16}}$ and all $b_i \in \mathbb{F}_2$. The polynomial $F = x^2 + x + \beta \in \mathbb{F}_{2^{16}}[x]$ is irreducible over $\mathbb{F}_{2^{32}}$, and G is chosen so that $F(\alpha) = 0$. Hence we may represent elements of $\mathbb{F}_{2^{32}}$ as $A_1\alpha + A_0$, with $A_0, A_1 \in \mathbb{F}_{2^{16}}$. We call this *representation B*. The product with another element $B_1\alpha + B_0$ of $\mathbb{F}_{2^{32}}$ can be computed as

$$(A_1\alpha + A_0)(B_1\alpha + B_0) = ((A_0 + A_1)(B_0 + B_1) + A_0B_0)\alpha + (A_0B_0 + A_1B_1\beta),$$

using 3 multiplications in $\mathbb{F}_{2^{16}}$ (the multiplication of A_1B_1 by β is just a shift in the polynomial basis representation of $\mathbb{F}_{2^{16}}$).

If we want to multiply two polynomials $a, b \in \mathbb{F}_2[x]$ using multipoint evaluation and interpolation at linear subspaces of one of the three fields $\mathbb{F}_{2^m}$ with $m \in \{16, 20, 32\}$ as above, we write a and b as

$$a = \sum_{0 \le i < r} a_i y^i, \quad b = \sum_{0 \le i < r} b_i y^i,$$

with $a_i, b_i \in \mathbb{F}_2[x]$ of degree less than $\frac{m}{2}$ and $y = x^{m/2}$. Then we regard y as a new indeterminate, substitute a generator γ of $\mathbb{F}_{2^m} = \mathbb{F}_2[\gamma]$ over $\mathbb{F}_2$ for x in the a_i and b_i, and multiply the resulting polynomials over $\mathbb{F}_{2^m}[y]$, as in section 2. Finally, we replace γ by x in the coefficients of the product polynomial, and compute $ab \in \mathbb{F}_2[x]$ by substituting $x^{m/2}$ for y. In this way, we can multiply polynomials in $\mathbb{F}_2[x]$ of degree less than $m2^{m-2}$.

In the case $m = 32$, where $\gamma = \alpha$, we have to convert the coefficients of the polynomials over $\mathbb{F}_{2^{32}}$ from representation A to representation B in order to perform multiplication in $\mathbb{F}_{2^{32}}$ as described above. This is done once when converting the polynomials over $\mathbb{F}_2$ to polynomials over $\mathbb{F}_{2^{32}}$, and the reverse conversion from representation B to representation A is done before computing $ab \in \mathbb{F}_2[x]$ from the product polynomial over $\mathbb{F}_{2^{32}}[y]$. The conversions are done using table lookups, and each of the two conversion tables is of size 256k bytes.

n	classical	K & O	Cantor $m=16$	Cantor $m=32$	§3 $m=20$
16384	0.25	0.04	0.08	0.08	0.32
32768	0.97	0.13	0.16	0.17	0.72
65536	3.87	0.39	0.33	0.36	1.60
131072	15.48	1.22	0.72	0.76	3.53
262144	61.97	3.55	1.59	1.66	7.48
524288	247.65	10.61		3.65	16.12
1048576	1002.73	31.96		8.00	35.46

Table 10.1: Average times in CPU seconds for one multiplication of two polynomials of degree $n - 1$.

Table 10.1 shows the average time in CPU seconds to multiply polynomials over $\mathbb{F}_2$ with the various algorithms for 10 pseudorandomly chosen inputs. There are no entries for Cantor's algorithm with $m = 16$ for degrees larger than 262144 because this is the maximal degree for which the method works (see above). As the theory predicts, our algorithm is slower than both variants of Cantor's algorithm. The main reason is that the polynomials s_i have coefficients in $\mathbb{F}_2$ in Cantor's algorithm, while their coefficients are in $\mathbb{F}_{2^{20}}$ in our algorithm. It is interesting that our implementation of Cantor's algorithm with $m = 32$ is nearly as fast as the variant with $m = 16$ (on a different but slower machine, it is even slightly faster). A possible reason is that the degree in y of the polynomials over $\mathbb{F}_{2^{16}}$ and $\mathbb{F}_{2^{32}}$ differs by a factor of 2 for the same polynomial in $\mathbb{F}_2$, so that the recursion depth in the multipoint evaluation and interpolation algorithms over $\mathbb{F}_{2^{32}}$ is one less than over $\mathbb{F}_{2^{16}}$; this seems to outweigh the fact that the cost for a multiplication in $\mathbb{F}_{2^{32}}$ is approximately three times the cost of a multiplication in $\mathbb{F}_{2^{16}}$. Since the differences in the running times are rather small and the variant with $m = 16$ is not applicable for degrees larger than 262144, we use $m = 32$ throughout.

Our implementation is about 3.3 times faster than the implementation of Montgomery (1991), which is presumably the consequence of higher processor speed, and about as fast as the implementation of Reischert (1995), whose program ran on a slower Sun Sparc 10/41 machine. The crossover point in our implementation between the classical algorithm and Karatsuba & Ofman is near degree 576, between Karatsuba & Ofman and Cantor with $m = 32$ near degree 35840.

One problem that often occurs when multiplying polynomials with the evaluation / interpolation method is that the running time of the resulting algorithm is rather discontinuous. In our situation, the running time function is nearly constant for polynomials with degrees between two successive powers of two and has discontinuities at the powers of two, both in Cantor's original method and ours. This is due to the fact that in order to be fast, we only evaluate and interpolate at point sets whose cardinality is a power of two. To smooth this behaviour, we use the following technique, also used by Reischert (1995). We note that the same method also applies to FFT based multiplication.

Multiplication of two polynomials $f, g \in \mathbb{F}_{q^m}[x]$ modulo s_i for some $i \in \{0, \ldots, m\}$ using Algorithm 2.4 and Algorithm 2.7 can be viewed as follows.

1. (Evaluation) Compute $f \bmod (x - \alpha)$ and $g \bmod (x - \alpha)$ for all $\alpha \in W_i$.

2. (Pointwise multiplication) Compute $fg \bmod (x - \alpha)$ for all $\alpha \in W_i$.

3. (Interpolation) Recover $fg \bmod s_i$ by Chinese remaindering.

The idea is now to replace x by x^l for some small $l \in \mathbb{N}$:

1. Compute $f \bmod (x^l - \alpha)$ and $g \bmod (x^l - \alpha)$ for all $\alpha \in W_i$.

2. Compute $fg \bmod (x^l - \alpha)$ for all $\alpha \in W_i$.

3. Recover $fg \bmod s_i(x^l)$ by Chinese remaindering.

With minor modifications, Algorithm 2.4 and Algorithm 2.7 can perform step 1 and step 3, respectively. Step 2 consists of q^i polynomial multiplications modulo degree l binomials, which are simply multiplications in $\mathbb{F}_{q^m}$ when $l = 1$. For $l \geq 2$, this can be done by multiplying q^i polynomials of degree less than l, e.g., using classical multiplication or Karatsuba & Ofman (one might even use the method recursively for larger l as in Reischert (1995)), and q^i subsequent divisions with remainder by degree l binomials, each using only $O(l)$ operations in $\mathbb{F}_{q^m}$.

	i	l
$\frac{1}{2}2^b < n \leq \frac{9}{16}2^b$	$b-4$	9
$\frac{9}{16}2^b < n \leq \frac{5}{8}2^b$	$b-3$	5
$\frac{5}{8}2^b < n \leq \frac{3}{4}2^b$	$b-3$	6
$\frac{3}{4}2^b < n \leq \frac{7}{8}2^b$	$b-3$	7
$\frac{7}{8}2^b < n \leq 2^b$	b	1

Table 10.2: Optimal values for i and l when multiplying polynomials of degree less than n over $\mathbb{F}_{2^{32}}$ with $b = \lceil \log_2 n \rceil \geq 4$.

The advantage of this is that we now have two parameters i and l instead of one, which can be chosen in such a way that the degree of the product polynomial fg is very close to the degree lq^i of the modulus $s_i(x^l)$. We have implemented the above algorithm in the case $q = 2$ and $m = 32$ for values of l between 2 and 16, with both classical and Karatsuba & Ofman type multiplication in step 2. Experiments showed that classical multiplication is no slower than Karatsuba & Ofman for such small values of l. Table 10.2 shows the optimal values for i and l that are suggested by measurements, and Figure 10.3 illustrates the improved running times of the variant in comparison to our implementation of Cantor's original algorithm and of Karatsuba & Ofman's algorithm.

Polynomial division. For division with remainder, we use the classical method for small degrees and Newton inversion (see Aho *et al.* 1974, Chapter 8) for large degrees. In the context of polynomial factorization, we are often in the situation that the divisor polynomial f is fixed throughout many divisions, namely the polynomial to be factored. Then Newton inversion admits the precomputation of

$$(x^{\deg f} \cdot f(x^{-1}))^{-1} \bmod x^{\deg f}, \tag{12}$$

which does not depend on the particular dividend, using $O(\mathsf{M}(\deg f))$ operations in $\mathbb{F}_2$, and the cost for computing one remainder modulo f is essentially the cost for two polynomial multiplications of degree less than $\deg f$. If we use an evaluation / interpolation scheme like Cantor's algorithm for polynomial multiplication,

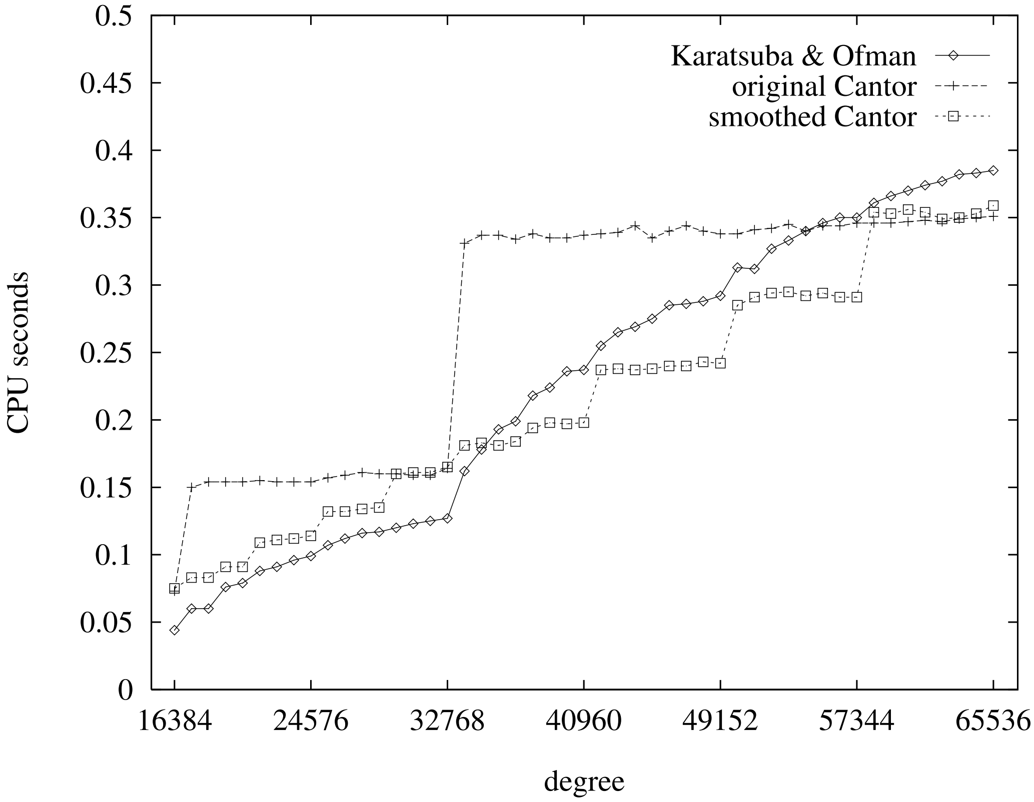

Figure 10.3: Running times of various polynomial multiplication algorithms.

further savings are possible by precomputing the multipoint evaluation of f and of the polynomial (12). This reduces the cost for one remainder computation modulo f to about $\frac{4}{3}$ of the cost for one polynomial multiplication of degree less than $\deg f$. A similar trick was used by Shoup (1995).

In our implementation of the irreducibility test described in section 9, we are often in the situation to compute many modular multiplications of the form $g \cdot h \bmod f$ where not only the divisor f but also one of the multiplicands, say h, is fixed. Using essentially the same trick as Shoup (1995), it is possible to reduce the cost for one such modular multiplication to about $\frac{4}{3}$ of the cost for one polynomial multiplication of degree less than $\deg f$ when using Cantor's algorithm for polynomial multiplication.

Table 10.4 shows the average time to compute one division with remainder using the classical method and Newton iteration, respectively, for 10 pseudorandomly chosen inputs. The crossover point between the two algorithms when the precomputation time is not counted is near degree 3584.

Polynomial gcds. For the computation of gcds, we use both the classical method and and a faster $O(\mathsf{M}(n) \log n)$ algorithm, also known as "half-gcd" (see

n	classical	Newton inversion	
		precomp.	remainder comp.
16384	0.24	0.07	0.09
32768	0.95	0.20	0.27
65536	3.80	0.80	0.51
131072	15.24	2.11	1.09
262144	61.06	4.95	2.38
524288	245.00	11.18	5.23
1048576	978.33	25.02	11.31

Table 10.4: Average times in CPU seconds for one division with remainder of a polynomial of degree $2n-3$ by a polynomial of degree $n-1$.

n	classical	"half-gcd"
16384	1.33	1.38
32768	5.79	3.66
65536	24.46	9.94
131072	100.56	26.85
262144	402.57	70.36
524288	1617.76	178.56
1048576	7017.88	439.81

Table 10.5: Average times in CPU seconds for one gcd of two polynomials of degree $n-1$.

Aho *et al.* 1974, Strassen 1983). Table 10.5 shows the average time in CPU seconds for the computation of one gcd using both methods for 10 pseudorandomly chosen inputs. The crossover point between the two algorithms is near degree 28672.

Polynomial factorization. Our polynomial factorization algorithm consists of the three stages described in section 5. For the squarefree factorization, we use Algorithm 5.1.

To compute the distinct degree factorization, we have implemented the coarse DDF algorithm as described in section 6, with early abort, and the interval partition defined by $c_j = 2j^2$, with intervals $\mathbf{I}_1 = \{1, 2\}$, $\mathbf{I}_2 = \{3, \ldots, 8\}$, $\mathbf{I}_3 = \{9, \ldots, 18\}$, ..., $\mathbf{I}_j = \{2(j-1)^2+1, \ldots, 2j^2\}$. Furthermore, we use a binary search fine DDF algorithm, as follows. To split a squarefree polynomial $f \in \mathbb{F}_2[x]$ whose irreducible factors have degrees in the interval $\{c+1, \ldots, d\} \subseteq \{1, \ldots, \deg f\}$, we cut the interval into two halves of approximately equal size, compute $a = \gcd(f, h) \in \mathbb{F}_2[x]$, where $h \in \mathbb{F}_2[x]$ is an interval polynomial for the lower half, and proceed recursively with a and f/a. The recursive process stops if $\deg f < 2(c+1)$, in which case f is irreducible, or otherwise if $c + 1 = d$. In the latter case, f is an equal-degree polynomial of order d, and we compute its equal degree factorization. As soon as the coarse DDF algorithm has detected that a fine DDF is necessary, the fine DDF algorithm can be executed in parallel to the coarse DDF algorithm, but we have only implemented a sequential version, where the fine DDF is performed before the coarse DDF algorithm proceeds.

As described in section 8, we reduce the number of modular multiplications in the computation of an interval polynomial for the interval $\{c+1, \ldots, d\}$ in the coarse DDF algorithm from $d - c + 1$ to about $\frac{d-c}{4}$. This saves a factor of $\frac{9}{20}$ of the running time, since the cost for a modular squaring is essentially the cost for two multiplications of polynomials of degree less than n, while a general modular multiplication uses three multiplications of polynomials of degree less than n. Montgomery (1991) uses a similar scheme with $\approx \frac{d-c}{3}$ modular multiplications.

We use the irreducibility test as described in section 9. The process is spawned as soon as the coarse DDF algorithm reaches degree 1000, and respawned every time the coarse DDF algorithm finds a new factor to check whether the remaining polynomial is irreducible.

The equal degree factorization is done as in Ben-Or (1981), at an expected cost of $O(d \cdot \mathsf{M}(rd) \log r)$ operations in $\mathbb{F}_2$ for a squarefree equal-degree polynomial of order d with r irreducible factors.

Factorization experiments. Table 10.6 shows examples of running times on the main machine for the factorization algorithm. The elapsed wall clock time differs from the CPU time on the main machine by at most three per cent in all experiments, and we have omitted it. The third column contains the amount of disk space in megabytes that the algorithm used for storing intermediate results. For the examples with degree 262143, the amount of disk space used was limited by the size of the hard disk. The fourth column shows the degree at which the coarse DDF algorithm ended or was aborted when the irreducibility test certified the remaining factor to be irreducible (in the latter case, the degree is written in italic), and the last column contains the *factorization pattern*, i.e., the degree sequence of the irreducible factors of the input polynomial. For example, in the fourth example of degree 65535, we have two different linear factors of multiplicity one each, one cubic factor of multiplicity 2, one factor of degree 42 and multiplicity one, and so on. Figure 10.7 illustrates the factorization patterns for the degree 16383 examples of Table 10.6. The largest factor is colored blue, the second largest green, etc. The width of a field corresponds to the degree of the respective irreducible factor. In our experiments, we never had to perform a fine DDF for total degrees above 8000 or an EDF for total degrees above 2. Since the algorithm is distributed over two machines, both the CPU time on the main machine and the elapsed wall clock time depend on the work load of both machines: e.g., if the load on the machine testing for irreducibility is high, the

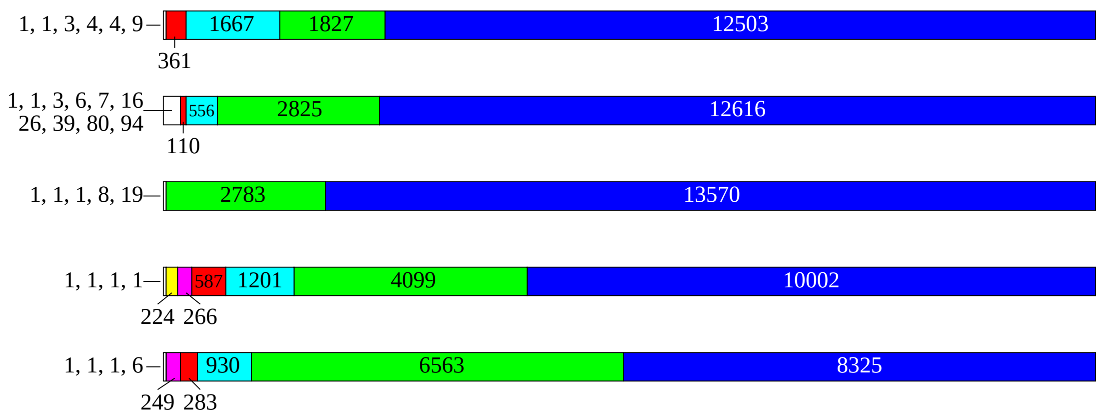

Figure 10.7: Factorization patterns for five pseudorandom polynomials of degree 16383 in $\mathbb{F}_2[x]$

abort degree of the coarse DDF algorithm (and hence the CPU time on the main machine) is higher than when the load is low.

We denote by $S_i(f)$ the degree of the ith largest irreducible factor of $f \in \mathbb{F}_2[x]$. The actual running time of the algorithm for an individual polynomial f depends on several factors:

- $S_1(f)$, the degree of the largest irreducible factor. With "early abort", the coarse DDF algorithm stops near the degree $\max\{S_2(f), \lfloor S_1(f)/2 \rfloor\}$ (because of blocking, the actual abort degree may be somewhat higher), so that the algorithm runs faster if the degrees of all irreducible factors are relatively small. However, for random polynomials this will rarely occur, since it is widely believed that there is some constant $\gamma \in \mathbb{R}$ with $0.5 < \gamma < 1$ such that $S_1(f)$ is $(\gamma + o(1))n$ on average for a random polynomial f of degree n, since a similar statement is true for integers (see Knuth & Trabb Pardo 1976).

- $S_2(f)$, the degree of the second largest irreducible factor of f. This is because the irreducibility test for the largest prime factor is launched when the coarse DDF algorithm reaches the degree $S_2(f)$.

Let $f \in \mathbb{F}_2[x]$ the polynomial to be factored, $n = \deg f$, and t the average time for the multiplication of two polynomials of degree about n in CPU seconds. Then the time used for one division with remainder of a polynomial of degree less than $2n$ by a factor of f is approximately $2t$ when using Karatsuba & Ofman's multiplication algorithm, or even about $\frac{4}{3}t$ when we use Cantor's method for multiplication. If $d \leq n/2$ is the degree where the coarse DDF algorithm stops and we neglect the cost for precomputations in the division algorithm, gcd computations, fine DDF, and EDF, the algorithm essentially performs d modular squarings and $d/4$ modular multiplications for the computation of the interval polynomials. This leads to an estimate for the total running time of $\approx \frac{11}{4}dt$ CPU seconds with Karatsuba & Ofman multiplication and $\approx \frac{23}{12}dt$ with Cantor's multiplication, which is in good accordance with the times in Tables 10.1 and 10.6 (the times in Table 10.6 are by a factor of up to 10 per cent higher than our estimate). The worst case for our DDF algorithm is when f has two irreducible factors of distinct degrees, both about $n/2$. The irreducibility test is of no help in this case, and we get an estimated running time of $\frac{11}{8}nt$ and $\frac{23}{24}nt$ CPU seconds, respectively.

Factoring trinomials can be done still faster, since division with remainder by a trinomial costs essentially the same as a polynomial addition. For example, the computation of an interval polynomial modulo a dense polynomial f for the interval $\{c+1,\ldots,d\}$ costs essentially $\frac{11}{4}(d-c)$ multiplications of polynomials of degree less than $\deg f$, while for a trinomial f this can be done with only about $\frac{1}{4}(d-c)$ multiplications. This leads to an estimated running time of $\frac{1}{4}dt$, where d and t are as above. We have implemented a variant of our factorization algorithm for trinomials of the form x^n+x+1 which exploits the sparseness, and factored the trinomial $x^{216091}+x+1$ from Montgomery (1991) in less than 7 hours of CPU time. We have also verified that our factors coincide with Montgomery's.

n	time	disk space	abort degree	factorization pattern
16383	$6'$	5	*2806*	1^5, 3, 4^2, 9, 361, 1667, 1827, 12503
16383	$8'$	7	*3698*	1^5, 3, 6, 7, 16, 26, 39, 80, 94, 110, 556, 2825, 12616
16383	$9'$	7	*3802*	1^2, 1, 8, 19, 2783, 13570
16383	$10'$	10	*4922*	1^4, 224, 266, 587, 1201, 4099, 10002
16383	$15'$	13	6728	1^3, 6, 24, 249, 283, 930, 6563, 8325
32767	$26'$	15	*3888*	1^2, 2^2, 8, 46, 306, 330, 32071
32767	$42'$	29	7442	1^2, 1^3, 2, 13, 23, 73, 140, 153, 393, 2145, 2177, 3308, 3695, 7245, 13395
32767	$56'$	42	10658	12, 30, 31, 34, 96, 1232, 1876, 3590, 3616, 10414, 11836
32767	$59'$	39	9839	1^2, 16, 22, 90, 102, 359, 791, 798, 1824, 9085, 19678
32767	$1^h05'$	40	10447	1^4, 2^2, 9, 55, 2141, 9659, 20895
65535	$1^h18'$	44	*5618*	1^2, 1^4, 2, 33, 143, 319, 551, 2772, 61709
65535	$1^h41'$	59	*7498*	1^3, 1, 10, 31, 590, 824, 1037, 1898, 3831, 57310
65535	$1^h43'$	60	*7634*	1^5, 6, 10, 11, 99, 653, 2355, 3364, 5413, 53619
65535	$2^h03'$	73	*9320*	1, 1, 3^2, 42, 71, 205, 607, 852, 2197, 3066, 3165, 7891, 47431
65535	$2^h14'$	79	*10082*	1^2, 5, 18, 29, 56, 80, 94, 259, 643, 1476, 3294, 8328, 51251
131071	$4^h26'$	138	*9094*	1^2, 1^2, 2, 14, 20, 23, 331, 1187, 3696, 125794
131071	$4^h41'$	146	*9356*	1, 1^2, 27, 164, 612, 5402, 124863
131071	$10^h10'$	331	*21184*	1, 1^3, 3, 449, 483, 1274, 18136, 110722
131071	$12^h18'$	428	27378	1, 1^4, 14, 67, 203, 631, 3546, 3580, 3877, 3924, 10400, 23894, 26057, 27069, 27804
131071	$13^h45'$	467	*29892*	1^2, 1^3, 2, 5, 8, 68, 111, 359, 1048, 1607, 12758, 15699, 28780, 70621
262143	$46^h35'$	1024	*47688*	1, 3, 215, 781, 16881, 29207, 29819, 43371, 45887, 95978
262143	$48^h25'$	1024	*48166*	2, 56, 110, 174, 1096, 1876, 13616, 29823, 44413, 170977

Table 10.6: CPU times for factoring some pseudorandomly chosen polynomials of degree n using two Sun Sparc Ultra 1 computers rated at 143 MHz.

References

A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading MA, 1974.

M. Ben-Or, Probabilistic algorithms in finite fields. In *Proc. 22nd IEEE Symp. Foundations Computer Science*, 1981, 394–398.

E. R. Berlekamp, Factoring polynomials over finite fields. *Bell System Tech. J.* **46** (1967), 1853–1859.

E. R. Berlekamp, Factoring polynomials over large finite fields. *Math. Comp.* **24** (1970), 713–735.

D. G. Cantor, On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A* **50** (1989), 285–300.

D. G. Cantor and H. Zassenhaus, A new algorithm for factoring polynomials over finite fields. *Math. Comp.* **36** (1981), 587–592.

C. Fiduccia, Polynomial evaluation via the division algorithm: the fast Fourier transform revisited. In *Proc. 4th Ann. ACM Symp. Theory of Computing*, Denver CO, 1972, 88–93.

P. Flajolet, X. Gourdon, and D. Panario, Random polynomials and polynomial factorization. Proc. ICALP '96, to appear, 1996.

P. Fleischmann and P. Roelse, Comparative implementations of Berlekamp's and Niederreiter's polynomial factorization algorithms. Preprint, 1995.

S. Gao and J. von zur Gathen, Berlekamp's and Niederreiter's polynomial factorization algorithms. In *Finite Fields: Theory, Applications and Algorithms*, ed. G. L. Mullen and P. J.-S. Shiue, vol. 168 of *Contemporary Mathematics.* Amer. Math. Soc., 1994, 101–115.

J. von zur Gathen and V. Shoup, Computing Frobenius maps and factoring polynomials. *Comput complexity* **2** (1992), 187–224.

J. von zur Gathen, X. Gourdon, and D. Panario, Average-case analysis of some polynomial factorization algorithms. Unpublished, 1995.

E. Kaltofen, Polynomial factorization 1987–1991. In *Proc. Latin'92, Lecture Notes in Computer Science* **583**, São Paulo, Brazil, 1992, 294–313.

E. Kaltofen and A. Lobo, Factoring high-degree polynomials by the black box Berlekamp algorithm. In *Proc. ISSAC '94*, ed. J. von zur Gathen and M. Giesbrecht. ACM Press, 1994, 90–98.

E. Kaltofen and V. Shoup, Subquadratic-time factoring of polynomials over finite fields. In *Proc. 27th Annual ACM Symp. Theory of Computing*. ACM Press, 1995, 398–406.

A. Karatsuba and Y. Ofman, Умножение многозначных чисел на автоматах. *Dokl. Akad. Nauk USSR* **145** (1962), 293–294. Multiplication of multidigit numbers on automata, Soviet Physics–Doklady **7** (1963), 595-596.

D. E. Knuth and L. Trabb Pardo, Analysis of a simple factorization algorithm. *Theoretical Computer Science* **3** (1976), 321–348.

R. Lidl and H. Niederreiter, *Finite Fields*, vol. 20 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading MA, 1983.

P. L. Montgomery, Factorization of $X^{216091} + X + 1 \bmod 2$ — a problem of Herb Doughty. Preprint, 1991.

H. Niederreiter, New deterministic factorization algorithms for polynomials over finite fields. *Contemporary Mathematics* **168** (1994), 251–268.

D. Reischert, Schnelle Multiplikation von Polynomen über GF(2) und Anwendungen. Diplomarbeit, University of Bonn, Germany, 1995.

A. Schönhage, Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Inf.* **7** (1977), 395–398.

V. Shoup, A new polynomial factorization algorithm and its implementation. To appear in J. Symb. Comp., 1995.

V. Strassen, The computational complexity of continued fractions. *SIAM J. Comput.* **12** (1983), 1–27.

D. Y. Y. Yun, On square-free decomposition algorithms. In *Proc. ACM Symp. Symbolic and Algebraic Computation*, ed. R. D. Jenks, 1976, 26–35.