MATHEMATICS OF COMPUTATION
Volume 71, Number 240, Pages 1677–1698
S 0025-5718(02)01421-7
Article electronically published on May 3, 2002

POLYNOMIAL FACTORIZATION OVER $\mathbb{F}_2$

JOACHIM VON ZUR GATHEN AND JÜRGEN GERHARD

ABSTRACT. We describe algorithms for polynomial factorization over the binary field $\mathbb{F}_2$, and their implementation. They allow polynomials of degree up to 250 000 to be factored in about one day of CPU time, distributing the work on two processors.

1. INTRODUCTION

The problem of polynomial factorization over the binary field $\mathbb{F}_2$ is, given a polynomial $f \in \mathbb{F}_2[x]$, to compute the factorization $f = f_1^{e_1} \cdots f_r^{e_r}$ with irreducible pairwise distinct polynomials $f_1, \ldots, f_r \in \mathbb{F}_2[x]$ and positive $e_1, \ldots, e_r$ in $\mathbb{N}$.

In the last 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). Many variants and asymptotically faster algorithms have been proposed more recently by von zur Gathen & Shoup (1992), Kaltofen (1992), Niederreiter (1994), Gao & von zur Gathen (1994), Kaltofen & Lobo (1994), and Kaltofen & Shoup (1997, 1998). Implementations are described in Montgomery (1991), Kaltofen & Lobo (1994), Shoup (1995), Fleischmann & Roelse (1996), and Roelse (1999). See von zur Gathen & Panario (2001) or Chapter 14 in von zur Gathen & Gerhard (1999) for surveys.

Section 2 gives an outline of the structure of some modern polynomial factorization algorithms. In Sections 3 and 4, 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 5 and 6, we indicate how the distinct degree factorization stage over $\mathbb{F}_2$ can be further speeded up by a special way of computing interval polynomials and by the use of an irreducibility test running in parallel on a second processor. An implementation of the polynomial factorization algorithm over $\mathbb{F}_2$ is described in Section 7, including examples of running times with pseudorandom inputs. It is able to factor pseudorandomly chosen polynomials of degree more than 250 000 in less than one day on two UltraSparc-II processors of a Sun Enterprise 450 clocked at 450 MHz. Recently, a parallelized variant of the software was able to factor a pseudorandom polynomial of degree more than

Received by the editor July 28, 2000.

2000 *Mathematics Subject Classification.* Primary 68W30; Secondary 11T06, 12Y05.

Key words and phrases. Fast algorithms, finite fields, Frobenius automorphism, polynomial factorization.

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

one million in four days of CPU time on a Linux PC with four Pentium III processors clocked at 500 MHz (Bonorden, von zur Gathen, Gerhard, Müller & Nöcker 2001).

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 optimized the equal degree factorization stage of our software.

A preliminary version of this work has appeared in Proc. ISSAC '96, ACM Press, Zürich, Switzerland, pp. 1-9. A more detailed version is in von zur Gathen & Gerhard (1996).

2. Polynomial factorization

Many of the modern polynomial factorization algorithms over finite fields (Cantor & Zassenhaus (1981), Ben-Or (1981), von zur Gathen & Shoup (1992), and Kaltofen & Shoup (1998), Algorithm D, but not those of Berlekamp (1967, 1970), Gao & von zur Gathen (1994), Kaltofen & Lobo (1994), Niederreiter (1994), and Kaltofen & Shoup (1998), Algorithm B) proceed in three stages:

1. **Squarefree factorization (SFF)**. Given a nonconstant 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
$$f = \prod_{1 \le i \le n} (g_i)^i.$$
2. **Distinct degree factorization (DDF)**. Given a nonconstant monic and squarefree polynomial $f \in \mathbb{F}_q[x]$ of degree n, compute its unique decomposition
$$f = \prod_{1 \le d \le n} h_d \tag{1}$$
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 $r \ge 2$ 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 at most $\mathsf{M}(n)$ operations in $\mathbb{F}_q$. By Schönhage & Strassen (1971) and Schönhage (1977), we may assume that $\mathsf{M}(n) \in O(n \log n \operatorname{loglog} n)$ (see also Cantor & Kaltofen 1991). In our implementation, we use $\mathsf{M}(n) \in O(n \log^{1.59} n)$ for $q = 2$ (Cantor 1989). 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., von zur Gathen & Gerhard 1999, Chapters 9 and 11).

Using the deterministic algorithm of Yun (1976), stage 1 can be performed at essentially the cost of one gcd, i.e., with $O(\mathsf{M}(n) \log n)$ operations in $\mathbb{F}_q$. The algorithm of Kaltofen & Shoup (1998) performs stage 2 using $O(n^{1.815} \log q)$ operations in $\mathbb{F}_q$. It is the asymptotically fastest of the currently known algorithms for distinct degree factorization when the field size q is fixed. 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 2.1. *Squarefree factorization.*
Input: *A nonzero polynomial* $f \in \mathbb{F}_2[x]$ *of degree* $n \in \mathbb{N}$.
Output: *Squarefree and pairwise coprime polynomials* $g_1, \ldots, g_n \in \mathbb{F}_2[x]$ *such that* $f = \prod_{1 \le i \le n} g_i^i$.

```
0. If $\deg f = 0$ then return the empty sequence.
1. Set $g = \gcd(f, f')$. { Comment: $g = (g_2 g_3)^2 (g_4 g_5)^4 (g_6 g_7)^6 \cdots$ }
2. Recursively compute the squarefree factorization $h_1, \ldots, h_m$ of $g^{1/2}$, where
   $2m = \deg g \le n$.
   { Comment: $h_i = g_{2i} g_{2i+1}$ for $1 \le i \le m$, where $g_{n+1} = 1$ is assumed. }
3. Set $h = f/g$. { Comment: $h = g_1 g_3 g_5 g_7 \cdots$ }
4. Repeat step 5 for $i$ from $m$ down to $1$.
5.         { Loop invariant: $h = \prod_{0 \le j \le i} g_{2j+1}$ }
           Set $g_{2i+1} = \gcd(h_i, h)$, $g_{2i} = h_i / g_{2i+1}$, and replace $h$ by $h / g_{2i+1}$.
6. Set $g_1 = h$ and $g_j = 1$ for $2m + 1 < j \le n$.
7. Return $g_1, \ldots, g_n$.
```

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 $\mathbb{F}_2$.

3. Distinct degree factorization

If one factors uniformly generated 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, Gourdon & Panario (1996) for a detailed analysis). This is confirmed by tests of our factorization routine on random inputs, as reported in Section 7. In this paper, we do not discuss the EDF further.

In the following, we briefly discuss the basic idea of all DDF algorithms over the finite field $\mathbb{F}_q$. In addition to $\mathsf{M}(n)$, 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 fixed polynomial of degree at most n.
- $\mathsf{Q}(n)$, the cost for one *modular qth power* of a polynomial of degree less than n modulo a fixed polynomial of degree at most n. Then $\mathsf{Q}(n) \le 2 \lfloor \log_2 q \rfloor \mathsf{P}(n)$. However, for $q = 2$, modular squaring is cheaper than a general modular multiplication.
- $\mathsf{D}(n)$, the cost for one *division with remainder* 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 functions above count operations in $\mathbb{F}_q$. We note that $\mathsf{P}(n), \mathsf{D}(n) \in O(\mathsf{M}(n))$, $\mathsf{Q}(n) \in O(\mathsf{M}(n) \log q)$, and $\mathsf{G}(n) \in O(\mathsf{M}(n) \log n)$. A high-level cost analysis in terms of these functions 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. In our algorithms, many modular products and modular squarings with the same fixed modulus occur, and we exploit this by preconditioning on the modulus as much as possible (see Section 7).

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 well-known algorithm for computing the DDF (1) of f by successively computing $\gcd(f, x^{q^i} - x)$ for $i = 1, 2, 3, \ldots, \lfloor n/2 \rfloor$ and removing it from f. Gauß states this method in a manuscript written in 1798 or 1799, but only published in his Nachlaß (Gauß 1863, p. 237). It was rediscovered several times; see Galois (1830), Serret (1866), Arwin (1918), and Cantor & Zassenhaus (1981). The cost of the algorithm is

$$\left\lfloor \frac{n}{2} \right\rfloor (\mathsf{Q}(n) + \mathsf{D}(n) + \mathsf{G}(n)) \in O(n \cdot \mathsf{M}(n)(\log q + \log n))$$

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 (Berlekamp 1984, exercise 3.6, see also Knopfmacher & Knopfmacher 1993). The cost for a gcd computation for polynomials of degree at most n differs by a factor of $O(\log n)$ from the cost for a multiplication or a division with remainder (in our experiments about $3 \log_2 n$), and for large values of n, the computation of a gcd is much more costly than a multiplication or a division with remainder, and computation time is wasted without noticeable progress.

To overcome this problem, some polynomial factorization algorithms (von zur Gathen & Shoup 1992, Kaltofen & Shoup 1998) 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*).

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 $1 \leq j \leq k$ are the *intervals* of the partition (see Figure 1). For $c, d \in \mathbb{N}$ with $c < d$, we define $(c, d]$ to be the set of polynomials

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

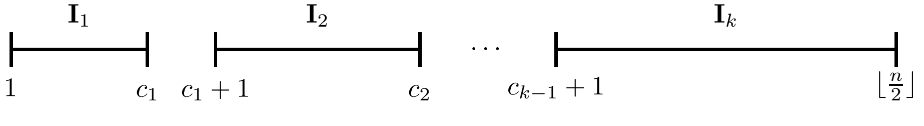

Figure 1. An interval partition.

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

$$\prod_{c_{j-1}<i\leq c_j} (x^{q^i} - x) \tag{2}$$

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

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

from Kaltofen & Shoup (1998). We note that interval polynomials need not be squarefree. Kaltofen & Shoup also use this terminology, but their interval polynomials differ from ours in that they are already reduced modulo the polynomial to be factored.

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 $a \operatorname{rem} b$. With the above notation, a coarse DDF algorithm can be stated as follows.

Algorithm 3.1. *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 the distinct–degree factors as in* (1)*, plus an irreducible factor of f of degree more than $n/2$, 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.     $B_j = \frac{B_{j-1}}{H_j}$.      { 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), Shoup (1995), and Kaltofen & Shoup (1998). Their algorithms differ in the computation of the interval polynomials. Note that the coarse DDF algorithm 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 (2) for I_j, step 3 can be implemented as follows.

(a) Set $I_j = 1$ and $a = x^{q^{c_{j-1}}} \operatorname{rem} B_{j-2}$, which has already been computed by former iterations of the loop 2 if $j \geq 2$.

(b) For $i = c_{j-1} + 1, \ldots, c_j$ replace a by $a^q \operatorname{rem} B_{j-1}$ and I_j by

$$I_j \cdot (a - x) \operatorname{rem} B_{j-1}.$$

Thus 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 the cost of the above algorithm is at most

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

operations in $\mathbb{F}_q$. Thus we have reduced the number of gcd computations from m to k in comparison to the simple DDF algorithm. 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 f. If $H_j \neq 1$ and the degree of H_j is less then $2(c_{j-1}+1)$, then 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 in order to keep things simple, we do not take this into account in the following analysis.

The interval partition $c_j = lj$ with constant interval sizes and $l = \lceil n^\beta \rceil$ is used by von zur Gathen & Shoup (1992) for $\beta = 1/2$ and by Kaltofen & Shoup (1998) for $0 \leq \beta \leq 1$. 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 (1998) is $O((n^\beta + n^{1-\beta})\mathsf{M}(n) \log n)$.

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

Let $m = \lfloor n/2 \rfloor$. 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 $d \geq 1$. For simplicity, we assume that $d \in \mathbb{N}$ in the following. 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 4.1. *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* (1), *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.     $b_i = \dfrac{b_{i-1}}{h_i}$.
       { Loop invariants: $H_j = b_i \prod_{c_{j-1} < l \leq i} h_l$ 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 (1998).

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 number of operations in $\mathbb{F}_q$ for a fine DDF on the interval $\mathbf{I}_j$ is $(c_j - c_{j-1})(\mathsf{Q}(n_j) + \mathsf{G}(n_j) + \mathsf{D}(n_j))$. Thus we obtain a total cost of

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

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 $\mathsf{D}(n)$ 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

$$\begin{aligned}&\frac{n}{2}\mathsf{P}(n) + \Big(dn^{(d-1)/d} + 2n^{1/d}\Big)\mathsf{D}(n) + \Big(\frac{n}{2} + dn^{(d-1)/d}\Big)\mathsf{Q}(n) \\ &\quad + \Big(dn^{(d-1)/d} + n^{1/d}\Big)\mathsf{G}(n)\end{aligned}$$

operations in $\mathbb{F}_q$ for n large enough. Minimizing the two exponents $(d-1)/d$ and $1/d$ leads to $d = 2$ and the following result.

Theorem 4.2. *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 the simple DDF algorithm, 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 (1998) 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, Gourdon & Panario (2002) 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 gcd computations.

5. 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 method has also been used by Montgomery (1991). We let $f \in \mathbb{F}_2[x]$ be 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 and d divisible by 4 for simplicity. Then instead of computing the remainder modulo f of the interval polynomial

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

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

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

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

$$\begin{aligned} V &= \prod_{u \in U} (y-u) = y^8 + v_4 y^4 + v_2 y^2 + v_1 y + v_0 \in \mathbb{F}_2[x,y], \\ U &= \{x, x^2, x^4, x^8, x+x^2+x^4, x+x^2+x^8, x+x^4+x^8, x^2+x^4+x^8\}, \end{aligned}$$

and $v_0, v_1, v_2, v_4 \in \mathbb{F}_2[x]$ are of degree at most 43. The polynomial V 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 $\mathbb{F}_2[x]$. Noting that

$$V(x, x^{2^{4i+3}}) \equiv x^{2^{4i+6}} + v_4 x^{2^{4i+5}} + v_2 x^{2^{4i+4}} + v_1 x^{2^{4i+3}} + v_0 \bmod f,$$

we compute $V(x, x^{2^{4i+3}}) \operatorname{rem} f$ using 4 modular squarings, 4 additions, and 3 modular multiplications by v_1, v_2, and v_4. Since the latter polynomials are "small", modular multiplications by them essentially cost the same as a scalar operation, and the dominant cost in the computation of $V(x, x^{2^{4i+3}}) \operatorname{rem} f$ is $4\mathsf{Q}(n)$. Thus the computation of the polynomial (4) modulo f costs essentially $(d-c)/4 \cdot \mathsf{P}(n) + (d-c)\mathsf{Q}(n)$ operations in $\mathbb{F}_2$, in contrast to $(d-c)\mathsf{P}(n) + (d-c)\mathsf{Q}(n)$ operations for the polynomial (3).

We note that the polynomial (4) 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 V, whose sole purpose is to make V a linearized polynomial (if V were not linearized, this would imply additional modular multiplications), may give rise to irreducible "phantom" factors in the gcd of f and (4) 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 never occurred in any of our experiments.

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

$$\begin{aligned}\prod_{c<2i\leq d}(x^{2^{2i}}+x^2)(x^{2^{2i}}+x) &= \prod_{c<2i\leq d}(x^{2^{2i-1}}+x)^2\cdot(x^{2^{2i}}+x)\\ &= \prod_{c<2i\leq d}(x^{2^{2i-1}}+x)\cdot\prod_{c<i\leq d}(x^{2^{i}}+x)\end{aligned}$$

by iterative modular multiplication with

$$(x^{2^{2i}}+x^2)(x^{2^{2i}}+x)=x^{2^{2i+1}}+(x^2+x)x^{2^{2i}}+x^3,$$

if c and d are both even. This takes essentially $(d-c)/2\cdot\mathsf{P}(n)+(d-c)\mathsf{Q}(n)$ operations in $\mathbb{F}_2$. Here, no phantom polynomials like in the coarse DDF algorithm can occur.

6. 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 7 indicate.

In our implementation, we run an irreducibility test on b in parallel to the coarse DDF algorithm on a second processor. If the coarse DDF algorithm finds a factor, the irreducibility test is aborted and restarted 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 about $n^{1/2}\times n^{1/2}$, 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. The asymptotic analysis of the irreducibility test does not show a substantial difference with the time 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 $x^{2^{\deg b}}\operatorname{rem} b$ and $x^{2^{\deg b/t}}\operatorname{rem} b$ 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 one precomputed value, namely for the largest binary prefix of e that is available, and then only have to shift and take care of the low order digits of e.

Algorithm 6.1. *Frobenius powers.*

Input: $b \in \mathbb{F}_2[x]$ *with* $b(0) \neq 0$ *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* 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. Take the binary representation $e = \sum_{0 \le j \le l} e_j 2^{l-j}$ of $e$, 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.
3. Repeat steps 4 and 5 for $j = l-k+1, \ldots, l$.
4.     Compute $h_j = g_{j-1}(g_{j-1}) \operatorname{rem} b$, using one modular composition.
5.     If $e_j = 0$ then set $g_j = h_j$ else compute $g_j = h_j^2 \operatorname{rem} b$.
       { Loop invariant: $g_j \equiv x^{2^{\lfloor e/2^{l-j} \rfloor}} \bmod b$ }
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 modular squarings is 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. In fact, in our application we are given $x^2, x^4, \ldots, x^{2^c}$ modulo some multiples b^* of b that divide the squarefree part of the polynomial f to be factored and such that b^*/b is not divisible by x. Then we have an additional division with remainder by b in step 2, whose cost turned out to be negligible as well in our experiments.

Since the space occupied by the data written to disk grows quadratically with the input size, the applicability of the above method is limited by the amount of disk space available. More precisely, if $\deg f = n$, then the table with the $c \le \lfloor n/2 \rfloor$ polynomials $x^2 \operatorname{rem} f, \ldots, x^{2^c} \operatorname{rem} f$ uses $c \lceil n/8 \rceil$ bytes of memory. For degree 262 143 and greater, the table size easily exceeds 1GB in our experiments, but it is easy to modify our approach so as to make maximum use of storage in that case.

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

7. Implementation and running times

In this section, we describe our implementation of the polynomial factorization algorithm over $\mathbb{F}_2$ on a Sun Enterprise 450 Model 4400 with four UltraSparc-II processors rated at 400 MHz each, running Solaris 5.6. Our algorithm uses at most two of the four processors. The software is a `C++` library called BiPolAr (for binary polynomial arithmetic), and the code was produced using the GNU compiler version 2.8.1. 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. The efficiency of the currently known polynomial factorization algorithms over finite fields relies on fast polynomial arithmetic, in particular, on fast polynomial multiplication. The multiplication method of Karatsuba (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. In contrast to Schönhage's algorithm, which evaluates and interpolates at suitable subgroups of the *multiplicative* group of an extension field $\mathbb{F}_{2^m}$ of degree $m \approx n^{1/2}$ over $\mathbb{F}_2$, a method by Cantor (1989) uses evaluation and interpolation at *additive* subgroups, i.e., $\mathbb{F}_2$-linear subspaces of $\mathbb{F}_{2^m}$, where $m \approx \log n$ is a power of two. It leads to an $O(n \log^{1.59} n)$ polynomial multiplication algorithm over $\mathbb{F}_2$. In von zur Gathen & Gerhard (1996), Cantor's approach is generalized (the extension degree m need no longer be a power of two), yielding an $O(n \log^2 n \,(\operatorname{loglog} n)^3)$ multiplication algorithm.

Reischert (1995) implemented several algorithms for polynomial multiplication over $\mathbb{F}_2$, including those by Karatsuba, Schönhage, and Cantor. Shoup (1995) 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. In his software package NTL (`http://www.shoup.net/ntl/`), Shoup also has a special data type and arithmetic for polynomials over $\mathbb{F}_2$. As of version 4.0a, NTL implements only Karatsuba's algorithm for polynomial multiplication over $\mathbb{F}_2$. Roelse (1999) reports on an implementation of both Karatsuba's and Cantor's algorithm.

Besides the school method, we have implemented the algorithm of Karatsuba and Cantor's method with subspaces of $\mathbb{F}_{2^{16}}$ and of $\mathbb{F}_{2^{32}}$. The timings in von zur Gathen & Gerhard (1996) indicate that the generalizations described there do not speed up the method. We did not implement Schönhage's algorithm either. 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 about 3/2.

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

Multiplication in $\mathbb{F}_{2^{16}}$ is implemented by means of two exponentiation and discrete logarithm tables 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. For multiplications in $\mathbb{F}_{2^{32}}$, the corresponding tables are far too large to fit in main memory, and we use an alternative approach.

We employ 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 what follows. The second is a polynomial basis representation of $\mathbb{F}_{2^{32}}$ over $\mathbb{F}_{2^{16}}$. We have chosen the primitive polynomial

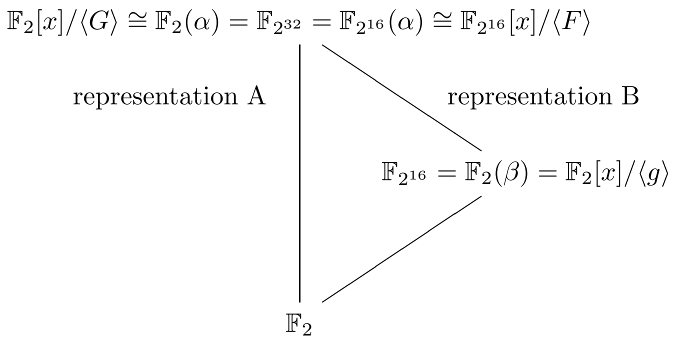
$\mathbb{F}_2[x]/\langle G\rangle \cong \mathbb{F}_2(\alpha) = \mathbb{F}_{2^{32}} = \mathbb{F}_{2^{16}}(\alpha) \cong \mathbb{F}_{2^{16}}[x]/\langle F\rangle$
representation A
representation B
$\mathbb{F}_{2^{16}} = \mathbb{F}_2(\beta) = \mathbb{F}_2[x]/\langle g\rangle$
$\mathbb{F}_2$

FIGURE 2. Two representations of $\mathbb{F}_{2^{32}}$.

$g = x^{16}+x^{15}+x^{13}+x^4+1 \in \mathbb{F}_2[x]$ 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 $\beta = x \bmod g \in \mathbb{F}_2[x]/\langle g\rangle = \mathbb{F}_{2^{16}}$ and all $b_i \in \mathbb{F}_2$. (There is no irreducible trinomial of degree 16 in $\mathbb{F}_2[x]$.) The polynomial $F = x^2+x+\beta \in \mathbb{F}_{2^{16}}[x]$ is irreducible over $\mathbb{F}_{2^{16}}$, and $G \in \mathbb{F}_2[x]$ was chosen as the minimal polynomial of a root $\alpha \in \mathbb{F}_{2^{32}}$ of F. Figure 2 illustrates the structure of the field extensions. 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

$$\begin{aligned}(A_1\alpha+A_0)(B_1\alpha+B_0) &= A_1B_1(\alpha+\beta)+(A_0B_1+A_1B_0)\alpha+A_0B_0\\ &= ((A_1+A_0)(B_1+B_0)+A_0B_0)\alpha+(A_1B_1\beta+A_0B_0),\end{aligned}$$

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 $\mathbb{F}_{2^{16}}$, followed by an addition of the generating polynomial g if necessary).

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 fields $\mathbb{F}_{2^m}$ with $m \in \{16, 32\}$ as above, we write a and b as

$$a = \sum_{0\le i<m/2} a_i y^i, \quad b = \sum_{0\le i<m/2} b_i y^i,$$

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 $\mathbb{F}_{2^m}[y]$. 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 $m2^{m-2}$, that is $2^{18} \approx 500\,000$ for $m = 16$ and $2^{35} \approx 32 \cdot 10^9$ for $m = 32$.

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.

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

n	classical	Karatsuba	Cantor $m=16$	Cantor $m=32$
16 384	0.07	0.01	0.02	0.02
32 768	0.28	0.04	0.04	0.04
65 536	1.11	0.11	0.09	0.08
131 072	4.47	0.33	0.19	0.18
262 144	17.85	1.00	0.43	0.40
524 288	71.38	3.00	0.93	0.88
1 048 576	285.58	9.02		1.89

Table 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 524 288 because of the method's degree constraint. It is interesting that our implementation of Cantor's algorithm with $m = 32$ is slightly faster than the variant with $m = 16$. 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 262 144, we use $m = 32$ throughout.

Our implementation is about 14 times faster than the implementation of Montgomery (1991) on a slower Sun 4/260 machine, about 4 times faster than the implementation of Reischert (1995) on a slower Sun Sparc 10/41 machine, and about 3.3 time faster than Roelse's (1999) implementation on a slower IBM RS6000 machine clocked at 67 MHz. The reason that our implementation is faster than the ones above is probably mainly due to a faster processor. We also ran a series of tests with NTL against our software on the Sun Enterprise 450, and it turned out that Shoup's implementation of Karatsuba's algorithm is faster than our Karatsuba implementation by a factor of about 1.6. Our implementation of Cantor's algorithm beats the NTL multiplication routine for degrees above 131 072. For example, for degree 1 048 576, our implementation is faster by a factor of about 3.

The crossover point in our implementation between the classical algorithm and Karatsuba is near degree 576, and between Karatsuba and Cantor with $m = 32$ near degree 35 840. We switch between the three multiplication algorithms at those degrees in our multiplication routine, which is also used as a subroutine when computing fast divisions with remainder and polynomial gcds.

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 for Cantor's algorithm is nearly constant for polynomials with degrees between two successive powers of two and has discontinuities at the powers of two. This is due to the fact that in order to be fast, we only evaluate and interpolate at point sets whose

TABLE 2. 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	Newton inversion	
		precomp.	remainder
16 384	0.07	0.02	0.03
32 768	0.28	0.06	0.07
65 536	1.11	0.27	0.09
131 072	4.51	0.65	0.19
262 144	17.99	1.47	0.42
524 288	71.93	3.25	0.91
1 048 576	288.44	7.49	1.96

cardinality is a power of two. We use a technique described in von zur Gathen & Gerhard (1996) to smooth this behavior.

Polynomial division. For division with remainder, we use the classical method for small degrees and Newton inversion (see von zur Gathen & Gerhard 1999, Chapter 9) 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

$$(x^{\deg f} \cdot f(x^{-1}))^{-1} \bmod x^{\deg f}, \tag{5}$$

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, further savings are possible by precomputing the multipoint evaluation of f and of the polynomial (5). This reduces the cost for one remainder computation modulo f to about $\mathsf{M}(\deg f)$. A similar trick was used by Shoup (1995).

One of the most important arithmetic operations in our implementation is modular squaring, i.e., the computation of $g^2 \bmod f$, where g is arbitrary and f the polynomial to be factored. Using the precomputed value (5) and a technique described in Reischert (1996), the cost for one modular squaring with fixed modulus f of degree n can be further reduced to about $\frac{5}{3}\,\mathsf{M}(n)$ when using the algorithm of Karatsuba for polynomial multiplication, and to about $\frac{5}{6}\,\mathsf{M}(n)$ when using Cantor's algorithm; see Reischert (1996) for the details.

In our implementation of the irreducibility test described in Section 6, we are often in the situation of having 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), we have reduced the cost for one such modular multiplication to about $\frac{4}{3}\,\mathsf{M}(\deg f)$ when using Cantor's algorithm for polynomial multiplication.

Table 2 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 neglecting the precomputation time is near degree 3584.

TABLE 3. Average times in CPU seconds for one gcd of two polynomials of degree $n-1$.

n	classical	"half-gcd"
16 384	0.45	0.43
32 768	2.01	1.12
65 536	8.44	2.96
131 072	36.56	7.69
262 144	154.60	19.38
524 288	586.16	47.67
1 048 576	2473.29	115.32

TABLE 4. Practice vs. theory: ratio of experimental running times and theoretical bounds. In the second and third columns, the ratio is some constant times the time from the last column in Tables 1 and 2, respectively, divided by $n(\log_2 n)^{\log_2 3}$. The ratio in the fourth column is some constant times the time from the last column in Table 3 divided by $n(\log_2 n)^{1+\log_2 3}$. The constants are chosen such that the ratios for $n = 65\,536$ are equal to 1.

n	Cantor $m = 32$	Newton inversion remainder	"half-gcd"
16 384	1.01	1.40	0.82
32 768	0.99	1.81	0.89
65 536	1.00	1.00	1.00
131 072	1.00	0.99	1.11
262 144	1.00	0.98	1.21
524 288	1.01	0.99	1.29
1 048 576	1.00	0.98	1.37

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 Aho, Hopcroft & Ullman 1974, Strassen 1983, or Chapter 11 in von zur Gathen & Gerhard 1999). Table 3 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 16 384.

Table 4 illustrates the correlation between actual running times and the theoretical prediction for multiplication, division with remainder, and gcd computation. We have normalized by the times at degree 65 536. Since our software is built with hybrid routines having different asymptotics in various ranges, these functions cannot be expected to be completely smooth.

Polynomial factorization. Our polynomial factorization algorithm consists of the three stages described in Section 2. For the squarefree factorization, we use Algorithm 2.1.

To compute the distinct degree factorization, we have implemented the coarse DDF algorithm as described in Section 3, with early abort, and the interval partition defined by $c_j = 2j^2$, with intervals $\mathbf{I}_1 = \{1,2\}$, $\mathbf{I}_2 = \{3,\ldots,8\}$, $\mathbf{I}_3 = \{9,\ldots,18\}$, $\ldots$, $\mathbf{I}_j = \{2(j-1)^2+1,\ldots,2j^2\}$. 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 5, we reduce the number of polynomial operations in the computation of an interval polynomial for the interval $\{c+1,\ldots,d\}$ in the coarse DDF algorithm from $d-c$ modular squarings and the same number of modular multiplications to $d-c$ modular squarings and about $(d-c)/4$ modular multiplications. E.g., when using Karatsuba type polynomial multiplication, this cuts the running time down by a factor of 27/56, since the cost for a modular squaring modulo a fixed polynomial f of degree n is $\approx \frac{5}{3}\,\mathsf{M}(n)$, while the cost for a general modular multiplication modulo f is $\approx 3\,\mathsf{M}(n)$. Montgomery (1999) uses a similar scheme with $\approx (d-c)/3$ modular multiplications.

We use the irreducibility test as described in Section 6. The process is spawned on a second processor as soon as the coarse DDF algorithm reaches degree 1000, and spawned again 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. This was necessary only for $d \leq 14$ in our experiments.

Factorization experiments. Tables 5 and 6 show examples of running times on the main processor for the factorization algorithm. The elapsed wall clock time differs from the CPU time on the main processor by less than one 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 technical reasons, we had to limit the maximal amount of disc space to 2GB. This was a real restriction only for degrees 262 143 and 524 287. 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). The benefit of using the irreducibility test may be considerable: for example, for the first polynomial of degree 131 071 in Table 6, the running time of our software without the irreducibility test is $4^h46'$, which is slower than the timing for the variant employing the irreducibility test by a factor of nearly 7. So even if we had run the irreducibility test in an interleaved fashion on the same processor, we would have obtained a speedup of about 3.5. 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 65 535, 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.

Table 5. CPU times and secondary storage for factoring some pseudorandomly chosen polynomials of degree n.

n	time	disk space	abort degree	factorization pattern
16 383	$92''$	5	2692	1^5, 3, 4^2, 9, 361, 1667, 1827, 12 503
16 383	$124''$	7	3676	1^5, 3, 6, 7, 16, 26, 39, 80, 94, 110, 556, 2825, 12 616
16 383	$130''$	7	3698	1^2, 1, 8, 19, 2783, 13 570
16 383	$147''$	9	4720	1^4, 224, 266, 587, 1201, 4099, 10 002
16 383	$221''$	13	6728	1^3, 6, 24, 249, 283, 930, 6563, 8325
32 767	$7'$	15	3854	1^2, 2^2, 8, 46, 306, 330, 32 071
32 767	$20'$	30	7442	1^2, 1^3, 2, 13, 23, 73, 140, 153, 393, 2145, 2177, 3308, 3695, 7245, 13 395
32 767	$14'$	42	10 658	12, 30, 31, 34, 96, 1232, 1876, 3590, 3616, 10 414, 11 836
32 767	$415'$	38	9839	1^2, 16, 22, 90, 102, 359, 791, 798, 1824, 9085, 19 678
32 767	16’	42	10 447	1^4, 2^2, 9, 55, 2141, 9659, 20 895
65 535	$14'$	37	4776	1^2, 1^4, 2, 33, 143, 319, 551, 2772, 61 709
65 535	$19'$	52	6602	1^3, 1, 10, 31, 590, 824, 1037, 1898, 3831, 57 310
65 535	$20'$	56	7098	1^5, 6, 10, 11, 99, 653, 2355, 3364, 5413, 53 619
65 535	$24'$	71	9082	1, 1, 3^2, 42, 71, 205, 607, 852, 2197, 3066, 3165, 7891, 47 431
65 535	$27'$	75	9610	1^2, 5, 18, 29, 56, 80, 94, 259, 643, 1476, 3294, 8328, 51 251

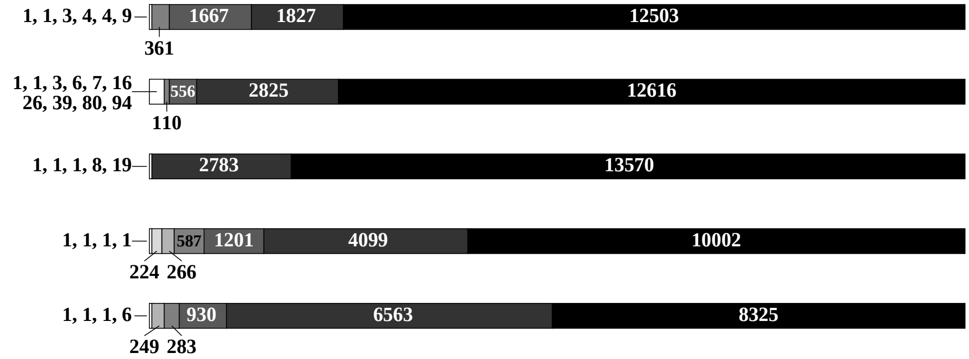

Figure 3. Factorization patterns for five pseudorandom polynomials of degree 16 383 in $\mathbb{F}_2[x]$.

Figure 3 illustrates the factorization patterns for the degree 16 383 examples of Table 5. 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 28. Since the algorithm is distributed over two processors, both the CPU time on the main processor and the elapsed

TABLE 6. CPU times and secondary storage for factoring some pseudorandomly chosen polynomials of degree n.

n	time	disk space	abort degree	factorization pattern
131 071	$41'$	121	7736	1^2, 1^2, 2, 14, 20, 23, 331, 1187, 3696, 125 794
131 071	$46'$	138	8824	1, 1^2, 27, 164, 612, 5402, 124 863
131 071	$1^h40'$	321	20 526	1, 1^3, 3, 449, 483, 1274, 18 136, 110 722
131 071	$2^h04'$	428	27 378	1, 1^4, 14, 67, 203, 631, 3546, 3580, 3877, 3924, 10 400, 23 894, 26 057, 27 069, 27 804
131 071	$2^h18'$	469	29 996	1^2, 1^3, 2, 5, 8, 68, 111, 359, 1048, 1607, 12 758, 15 699, 28 780, 70 621
262 143	$4^h14'$	744	23 794	1, 1, 9, 33, 41, 96, 291, 336, 795, 1860, 2906, 18 555, 237 219
262 143	$7^h43'$	1487	47 560	1, 3, 215, 781, 16 881, 29 207, 29 819, 43 371, 45 887, 95 978
262 143	$7^h59'$	1495	47 844	2, 56, 110, 174, 1096, 1876, 13 616, 29 823, 44 413, 170 977
262 143	$10^h40'$	2044	65 412	1, 2^2, 3, 11, 109, 259, 416, 1170, 1519, 1937, 2488, 3125, 7247, 33 587, 62 673, 147 594
262 143	$15^h47'$	2047	95 922	4, 7, 37, 96, 103, 177, 738, 1268, 1649, 1796, 6283, 7015, 95 459, 147 520
524 287	$16^h15'$	2046	42 839	1, 1^2, 20, 830, 1443, 1538, 2054, 3175, 33 369, 34 852, 447 003
524 287	$20^h17'$	2046	53 792	1, 1^2, 15, 41, 132, 188, 1097, 4480, 7436, 14 419, 17 159, 44 788, 434 529
524 287	$43^h48'$	2046	125 352	2, 12, 14, 14, 85, 113, 148, 296, 343, 345, 6338, 31 278, 48 200, 119 622, 317 477
524 287	$44^h06'$	2046	126 310	3, 25, 338, 1532, 12 564, 33 055, 98 748, 122 164, 255 858
524 287	$64^h25'$	2046	183 618	1, 1^3, 5, 52, 62, 67, 403, 561, 569, 1566, 1776, 20 384, 183 268, 315 570

wall clock time depend on the work load of both processors: e.g., if the load on the processor testing for irreducibility is high, the abort degree of the coarse DDF algorithm (and hence the CPU time on the main processor) is higher than when the load is low.

Let $f \in \mathbb{F}_2[x]$ be the polynomial to be factored and $n = \deg f$. We denote by S_i the degree of the ith largest irreducible factor of $f \in \mathbb{F}_2[x]$. The actual running time of the algorithm on input f mainly depends on S_2, for the following reasons. If $S_1 \leq 2S_2$, then the irreducibility test is of no help, and the coarse DDF algorithm with "early abort" stops at degree S_2 (due to the blocking, the actual abort degree is somewhat higher in our experiments). Otherwise, the irreducibility test is spawned for the last time when the coarse DDF algorithm reaches degree S_2, and in our experiments the remaining factor is certified to be irreducible quite soon afterwards. In fact, in Tables 5 and 6 the abort degree d never exceeds S_2 by more

TABLE 7. Practive vs. theory: ratio of experimental running times and theoretical bounds for factorization. The times are taken from Tables 5 and 6, and the constant c is chosen such that the first ratio in the row for $n = 65\,535$ is equal to 1.

n	$c \cdot \text{time}/(S_2\, n(\log_2 n)^{\log_2 3})$				
16 383	0.81	0.70	0.75	0.57	0.54
32 767	8.65	0.59	0.58	0.71	0.73
65 535	1.00	0.95	0.72	0.60	0.62
131 071	0.98	0.75	0.48	0.40	0.42
262 143	0.55	0.41	0.43	0.41	0.40
524 287	0.52	0.50	0.41	0.40	0.39

than 9004. We have $S_2 \le d \le 2.1 \cdot S_2$ in all our experiments, with one exception, namely for the first polynomial of degree 32 767, where S_2 is very small. In fact, the average of d/S_2 over 29 experiments (all but the exceptional one) is about 1.2.

The running time is essentially proportional to $S_2 \cdot t$, where t is the average time for the multiplication of two polynomials of degree about n. This is $O(n\,\mathsf{M}(n))$ or $O^{\sim}(n^2)$ when we use Cantor's multiplication algorithm, since $S_2 < n/2$. Similar to Table 4, Table 7 displays the ratios of actual running times versus predictions, normalizing at the first polynomial of degree 65 535. The exceptional polynomial mentioned before is reflected by the ratio 8.65 in Table 7.

The average times for one modular squaring and one general modular multiplication modulo a factor of f are at most $\approx \frac{5}{6}t$ and $2t$, respectively, when using Cantor's multiplication method. If $d \le n/2$ is the degree where the coarse DDF algorithm stops and if we neglect the cost for precomputations in the division algorithm, gcd computations, fine DDF, and EDF, then the algorithm essentially performs d modular squarings and about $d/4$ modular multiplications to compute interval polynomials. This yields an estimate for the total running time of $\approx \frac{4}{3}dt$ with Cantor's multiplication, which is in good accordance with the times in Tables 1 and 6. In fact, the times in Table 6 are higher by a factor of up to about 20 per cent, due to the gcd computations. The worst case for our DDF algorithm is when f has two irreducible factors of distinct degrees, both about $n/2$. The irreducibility test does not help then, and we get an estimated running time of $\frac{2}{3}nt$.

Factoring trinomials can be done still faster, since division with remainder by a trinomial costs essentially the same as a polynomial addition. For example, when using Cantor's multiplication, the cost for the computation of an interval polynomial modulo a dense polynomial f for the interval $\{c+1, \ldots, d\}$ is essentially $\frac{4}{3}(d-c)\mathsf{M}(\deg f)$, while for a trinomial f this can be done with only about $(d-c)/4 \cdot \mathsf{M}(\deg f)$ operations. 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 $x^{216\,091} + x + 1$ from Montgomery (1991), whose software was able to factor it in about 45 hours on a Sun 4/260, in about one and a half hours of CPU time. We have also verified that our factors coincide with Montgomery's.

Roelse (1999) states a parallel running time of about 10 hours for factoring a random polynomial in $\mathbb{F}_2[x]$ of degree 300 000 by Niederreiter's algorithm, using 256

IBM RS6000 processors running at 67 MHz each. The largest irreducible factor of this polynomial has degree 126 929, and the second largest one has degree 100 948. Thus our irreducibility test would be of no help here. Extrapolating our running times for the last polynomial of degree 262 143 in Table 6, we conjecture that we would be able to factor Roelse's polynomial in about 20 hours, using one 400 MHz UltraSparc-II processor. While the linear-algebra based algorithm of Niederreiter, which Roelse implements, appears to be better suited for parallelization than our algorithm, the sequential running time of our algorithm is only $O^{\sim}(n^2)$ for polynomials of degree n, in contrast to $O(n^3)$ for Niederreiter's algorithm. Moreover, when we do not use the irreducibility test, then the storage requirement for our algorithm is $O(n)$, while Niederreiter's algorithm uses $O(n^2)$.

We used NTL to verify for all our factorizations that the product of all found factors equals the input polynomial. Moreover, we checked with the irreducibility test of NTL that all factors of degree less than 150 000 in our factorizations are indeed irreducible.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading MA, 1974. MR **54:**1706

[2] A. Arwin, *Über Kongruenzen von dem fünften und höheren Graden nach einem Primzahlmodulus*, Ark. Mat. **14** (1918), no. 7, 1–46.

[3] M. Ben-Or, *Probabilistic algorithms in finite fields*, Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computer Science, Nashville TN, 1981, pp. 394–398.

[4] E. R. Berlekamp, *Factoring polynomials over finite fields*, Bell System Technical Journal **46** (1967), 1853–1859. MR **36:**2314

[5] ______, *Factoring polynomials over large finite fields*, Math. Comp. **24** (1970), no. 11, 713–735. MR **43:**1948

[6] ______, *Algebraic coding theory*, Aegean Park Press, 1984. First edition McGraw Hill, New York, 1968. MR **38:**6873

[7] Olaf Bonorden, Joachim von zur Gathen, Jürgen Gerhard, Olaf Müller, and Michael Nöcker, *Factoring a binary polynomial of degree over one million*, ACM SIGSAM Bull. **35** (1), 2001, 16–18.

[8] David G. Cantor, *On arithmetical algorithms over finite fields*, J. Combin. Theory Ser. A **50** (1989), 285–300. MR **90f:**11100

[9] David G. Cantor and Erich Kaltofen, *On fast multiplication of polynomials over arbitrary algebras*, Acta Inform. **28** (1991), 693–701. MR **92i:**68068

[10] David G. Cantor and Hans Zassenhaus, *A new algorithm for factoring polynomials over finite fields*, Math. Comp. **36** (1981), no. 154, 587–592. MR **82e:**12020

[11] Philippe Flajolet, Xavier Gourdon, and Daniel Panario, *Random polynomials and polynomial factorization*, Proceedings of the 23rd International Colloquium on Automata, Languages and Programming ICALP 1996, Paderborn, Germany (F. Meyer auf der Heide and B. Monien, eds.), Lecture Notes in Comput. Sci., no. 1099, Springer-Verlag, 1996, pp. 232–243. INRIA Rapport de Recherche No 3370, March 1998, 28 pages. MR **98e:**68123

[12] Peter Fleischmann and Peter Roelse, *Comparative implementations of Berlekamp's and Niederreiter's polynomial factorization algorithms*, Finite Fields and Applications, Proceedings of the third international conference, Glasgow, UK, July 1995 (S. Cohen and H. Niederreiter, eds.), London Math. Soc. Lecture Notes Ser., no. 233, Cambridge University Press, 1996, pp. 73–83. MR **98a:**12009

[13] É. Galois, *Sur la théorie des nombres*, Bulletin des sciences mathématiques Férussac **13** (1830), 428–435. See also *Journal de Mathématiques Pures et Appliquées* **11** (1846), 398–407, and *Écrits et mémoires d'Évariste Galois* (Robert Bourgne and J.-P. Azra, eds.), Gauthier-Villars, Paris, 1962, 112–128. MR **27:**21

[14] Shuhong Gao and Joachim von zur Gathen, *Berlekamp's and Niederreiter's polynomial factorization algorithms*, Finite Fields: Theory, Applications and Algorithms (G. L. Mullen

and P. J.-S. Shiue, eds.), Contemp. Math., no. 168, American Mathematical Society, 1994, pp. 101–115. MR **95f:**11106

[15] Joachim von zur Gathen and Jürgen Gerhard, *Arithmetic and factorization of polynomials over $\mathbb{Z}_2$*, Tech. Report tr-rsfb-96-018, University of Paderborn, Germany, 1996, 43 pages.

[16] ______, *Modern computer algebra*, Cambridge University Press, Cambridge, UK, 1999. MR **2000j:**68205

[17] J. von zur Gathen, X. Gourdon, and D. Panario, *Average cost of baby-step/giant-step polynomial factorization algorithms*, in preparation, 2002.

[18] Joachim von zur Gathen and Daniel Panario, *Factoring polynomials over finite fields: A survey*, J. Symbolic Comput. **31** (2001), no. 1–2, 3–17.CMP 2001:07

[19] Joachim von zur Gathen and Victor Shoup, *Computing Frobenius maps and factoring polynomials*, Comput. Complexity **2** (1992), 187–224. MR **94d:**12011

[20] Carl Friedrich Gauß, *Theoria residuorum biquadraticorum, commentatio secunda*, Werke **II**, Königliche Gesellschaft der Wissenschaften, Göttingen, 1863, reprinted by Georg Olms Verlag, Hildesheim New York, 1973, pp. 93–150. MR **82e:**01022

[21] E. Kaltofen, *Polynomial factorization 1987–1991*, Proceedings of LATIN '92, São Paulo, Brazil (I. Simon, ed.), Lecture Notes in Comput. Sci., no. 583, Springer-Verlag, 1992, pp. 294–313. MR **94f:**68019

[22] E. Kaltofen and A. Lobo, *Factoring high-degree polynomials by the black box Berlekamp algorithm*, Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation ISSAC '94, Oxford, UK (J. von zur Gathen and M. Giesbrecht, eds.), ACM Press, 1994, pp. 90–98.

[23] Erich Kaltofen and Victor Shoup, *Fast polynomial factorization over high algebraic extensions of finite fields*, Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation ISSAC '97, Maui HI (Wolfgang W. Küchlin, ed.), ACM Press, 1997, pp. 184–188. MR **2000i:**68174

[24] ______, *Subquadratic-time factoring of polynomials over finite fields*, Math. Comp. **67** (1998), no. 223, 1179–1197, Extended Abstract in Proceedings of the Twenty-seventh Annual ACM Symposium on the Theory of Computing, Las Vegas NV, ACM Press, 1995, 398–406. MR **99m:**68097

[25] A. Karatsuba and Yu. Ofman, Умножение многозначных чисел на автоматах, Доклады Академий Наук СССР **145** (1962), 293–294. A. Karatsuba and Yu. Ofman, Multiplication of multidigit numbers on automata, Soviet Physics–Doklady **7** (1963), 595–596.

[26] Arnold Knopfmacher and John Knopfmacher, *Counting irreducible factors of polynomials over a finite field*, Discrete Math. **112** (1993), 103–118. MR **94a:**11188

[27] Rudolf Lidl and Harald Niederreiter, *Finite fields*, Encyclopedia Math. Appl., no. 20, Addison-Wesley, Reading MA, 1983. MR **86c:**11106

[28] Peter L. Montgomery, *Factorization of $X^{216091}+X+1$ mod 2—A problem of Herb Doughty*, manuscript, February 1991.

[29] Harald Niederreiter, *New deterministic factorization algorithms for polynomials over finite fields*, Finite fields: theory, applications and algorithms (G. L. Mullen and P. J.-S. Shiue, eds.), Contemp. Math., no. 168, American Mathematical Society, 1994, pp. 251–268. MR **95f:**11100

[30] Daniel Reischert, *Schnelle Multiplikation von Polynomen über GF(2) und Anwendungen*, Diplomarbeit, Institut für Informatik II, Rheinische Friedrich-Wilhelm-Universität Bonn, Germany, August 1995.

[31] ______, *Multiplication by a square is cheap over $\mathbb{F}_2$*, manuscript, 1996.

[32] Peter Roelse, *Factoring high-degree polynomials over F_2 with Niederreiter's algorithm on the IBM SP2*, Math. Comp. **68** (1999), no. 226, 869–880. MR **99i:**11112

[33] A. Schönhage, *Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2*, Acta Inform. **7** (1977), 395–398. MR **55:**9604

[34] A. Schönhage and V. Strassen, *Schnelle Multiplikation großer Zahlen*, Computing **7** (1971), 281–292. MR **45:**1431

[35] J.-A. Serret, *Cours d'algèbre supérieure*, 3rd ed., Gauthier-Villars, Paris, 1866.

[36] Victor Shoup, *A new polynomial factorization algorithm and its implementation*, J. Symbolic Comput. **20** (1995), 363–397. MR **97d:**12011

[37] V. Strassen, *The computational complexity of continued fractions*, SIAM J. Comput. **12** (1983), no. 1, 1–27. MR **84b:**12004

[38] David Y. Y. Yun, *On square-free decomposition algorithms*, Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation SYMSAC '76, Yorktown Heights NY (R. D. Jenks, ed.), ACM Press, 1976, pp. 26–35.

Fachbereich 17 Mathematik-Informatik, Universität Paderborn, D-33095 Paderborn, Germany
E-mail address: `gathen@upb.de`

Fachbereich 17 Mathematik-Informatik, Universität Paderborn, D-33095 Paderborn, Germany
E-mail address: `jngerhar@upb.de`