COMPUTING FROBENIUS MAPS AND FACTORING POLYNOMIALS

Joachim von zur Gathen and Victor Shoup

Abstract. A new probabilistic algorithm for factoring univariate polynomials over finite fields is presented. To factor a polynomial of degree n over $\mathbb{F}_q$, the number of arithmetic operations in $\mathbb{F}_q$ is $O((n^2+n\log q)\cdot(\log n)^2 \operatorname{loglog} n)$. The main technical innovation is a new way to compute Frobenius and trace maps in the ring of polynomials modulo the polynomial to be factored.

Subject classifications. 68Q40; 11Y16, 12Y05.

1. Introduction

We consider the problem of factoring a univariate polynomial over a finite field.

This problem plays a central rôle in computational algebra. Indeed, many of the efficient algorithms for factoring univariate and multivariate polynomials over finite fields, the field of rational numbers, and finite extensions of the rationals solve as a subproblem the problem of factoring univariate polynomials over finite fields (Kaltofen 1990). This problem also has important applications in number theory (Buchmann 1990), coding theory (Berlekamp 1968), and cryptography (Odlyzko 1985).

In this paper, we describe a new algorithm for this problem whose asymptotic running time improves upon previous results. Our main result is a probabilistic algorithm for factoring a polynomial of degree n over a finite field $\mathbb{F}_q$ with q elements that uses $O((n^2+n\log q)\cdot(\log n)^2 \operatorname{loglog} n)$ operations in $\mathbb{F}_q$ (additions, subtractions, multiplications, divisions, and zero tests). This can be expressed more briefly, if less precisely, as $O\tilde{~}(n^2+n\log q)$, where the "Soft-O" notation introduced in von zur Gathen (1985) and Babai *et al.* (1988) is used to suppress logarithmic factors: $g = O\tilde{~}(h)$ means that $g = O(h(\log h)^k)$ for some constant k.

Previously, the asymptotically fastest algorithms were due to Berlekamp (1970), Cantor and Zassenhaus (1981), and Ben-Or (1981). These algorithms are also probabilistic. Berlekamp's algorithm can be implemented with $O\tilde{~}(n^\omega +$

$n \log q)$ operations in $\mathbb{F}_q$, if we can multiply two $n \times n$ matrices with $O(n^\omega)$ operations; we can choose $\omega < 2.376$ (Coppersmith & Winograd 1990). The Cantor-Zassenhaus algorithm takes $O\tilde{}(n^2 \log q)$ operations in $\mathbb{F}_q$, and Ben-Or's algorithm also has this running time.

The contrast between the running time of our algorithm and these other algorithms is most striking when $\log q \approx n$. In this case, our algorithm takes $O\tilde{}(n^2)$ operations in $\mathbb{F}_q$, Berlekamp's takes $O\tilde{}(n^\omega)$, and the Cantor-Zassenhaus algorithm takes $O\tilde{}(n^3)$.

Our algorithm is a variant of the Cantor-Zassenhaus algorithm, and, like that algorithm, breaks the general factoring factoring problem into three subproblems:

1. *squarefree factorization*, i.e., making an arbitrary polynomial squarefree;
2. *distinct-degree factorization*, i.e., splitting a squarefree polynomial into polynomials whose irreducible factors have the same degree;
3. *equal-degree factorization*, i.e., completely factoring a squarefree polynomial whose irreducible factors have the same degree.

The algorithm of Yun (1976) computes the squarefree factorization with $O\tilde{}(n \log q)$ operations in $\mathbb{F}_q$.

Let $f \in \mathbb{F}_q[x]$ be the polynomial to be factored, $n = \deg f$, and $R = \mathbb{F}_q[x]/(f)$. We shall always use the symbol ξ to denote $(x \bmod f) \in R$. In solving the distinct-degree and equal-degree factorization problems, the *Frobenius map* on R, which sends $\alpha \in R$ to α^q, plays a fundamental rôle. Several algorithms for these problems compute some of the iterates $\alpha, \alpha^q, \ldots, \alpha^{q^n}$ of the Frobenius map for various α in R.

At the heart of our algorithms is the following idea, which we call the *polynomial representation* of the Frobenius map. Suppose we have precomputed the special element $\beta = \xi^q \in R$. An arbitrary $\alpha \in R$ can be represented as $\alpha = (g \bmod f) \in R$, where $g \in \mathbb{F}_q[x]$ has degree less than n. Thus $\alpha = g(\xi)$, and the problem of computing α^q can be viewed as the problem of evaluating the polynomial g at the point β, since

$$\alpha^q = g(\xi)^q = g(\xi^q) = g(\beta).$$

This polynomial representation of the Frobenius map is due to Erich Kaltofen who used it to design a factoring algorithm that takes $O\tilde{}(n^3 + n \log q)$ operations in $\mathbb{F}_q$ and only linear space. Previous algorithms with this running time used quadratic space.

Combining the polynomial representation of the Frobenius map with an algorithm for fast multi-point polynomial evaluation over the ring R allows us to compute $\alpha, \alpha^q, \ldots, \alpha^{q^n}$ for a given $\alpha \in R$ with $O\tilde{}(n^2 + n \log q)$ operations in $\mathbb{F}_q$, whereas the obvious repeated squaring algorithm uses $O\tilde{}(n^2 \log q)$ operations in $\mathbb{F}_q$, and an algorithm based on a matrix representation of the Frobenius map uses $O\tilde{}(n^\omega + n \log q)$ operations in $\mathbb{F}_q$. This new approach leads to our basic algorithms for solving the distinct-degree and equal-degree factorization problems, which use $O\tilde{}(n^2 + n \log q)$ operations in $\mathbb{F}_q$. These are described in Section 3; a part of the probabilistic analysis is done in Section 4.

A high-level view of the three approaches to computing the Frobenius map is that repeated squaring treats it as a multiplicative map on R, the matrix representation as an $\mathbb{F}_q$-linear function, and the polynomial representation as an $\mathbb{F}_q$-algebra endomorphism.

The running time of our algorithm is fairly insensitive to the largest degree d of irreducible factors of f. However, the algorithm of Cantor and Zassenhaus actually uses only $O\tilde{}(nd \log q)$ operations in $\mathbb{F}_q$, and hence may be faster than ours when d is small. In particular, our method does not improve root-finding.

Section 5 takes another look at equal-degree factorization. For this problem, we do not need to compute the individual iterates of the Frobenius map, but only sums of the form $\sum_{0 \le i < d} \alpha^{q^i}$, where d is the degree of the irreducible factors of f. We call such a sum a *trace map*, as it is given by the formula for the trace from $\mathbb{F}_{q^d}$ to $\mathbb{F}_q$. The polynomial representation of the Frobenius map allows us to compute the trace map for a given $\alpha \in R$ using only $O\tilde{}(n^{(\omega+1)/2} + n \log q)$ operations in $\mathbb{F}_q$. This leads to an algorithm for equal-degree factorization with the same time bound, which is $O\tilde{}(n^{1.7} + n \log q)$.

At the present time, we have the anomaly that our algorithm for the "easy" problem of distinct-degree factorization, which uses $O\tilde{}(n^2 + n \log q)$ operations in $\mathbb{F}_q$, is asymptotically slower than our algorithm for the "harder" problem of equal-degree factorization, which uses $O\tilde{}(n^{1.7} + n \log q)$ operations in $\mathbb{F}_q$. The latter problem seems harder since no deterministic polynomial time algorithm for it is known if the characteristic of the field is large. In Section 6, we discuss a special version of the multi-point polynomial evaluation problem that is the bottleneck of our current algorithms for distinct-degree factorization: asymptotic improvements on it would immediately improve the time for distinct-degree factorization, and hence the time for the general factoring problem. No lower bounds are known for the factoring problem, except that factoring quadratic polynomials takes $\Omega(\log q)$ operations in $\mathbb{F}_q$ (von zur Gathen & Seroussi 1991).

Throughout the paper, we also discuss the space requirements—in terms of storage of elements of $\mathbb{F}_q$—of our algorithms. Our basic algorithms in Section 3 use space $O(n^2)$. The equal-degree factoring algorithm in Section 5 uses space $O(n \log n)$, and we present in Section 6 a distinct-degree factorization algorithm with space $O(n^{3/2})$.

In Section 7, we show how to test polynomials for irreducibility with $O\tilde{}(n^{(\omega+1)/2} + n \log q)$ operations in $\mathbb{F}_q$. This improves upon the previous methods of Butler (1954), which takes $O\tilde{}(n^\omega + n \log q)$ operations in $\mathbb{F}_q$, and Rabin (1980), using $O\tilde{}(n^2 \log q)$ operations in $\mathbb{F}_q$. We obtain a similar improvement for finding normal bases, and—with a different method—an even faster algorithm for computing traces in extension fields.

In Section 8, we briefly discuss the usefulness of our methods under the restriction that so-called "classical" algorithms for polynomial and matrix arithmetic are used.

Our algorithms for distinct-degree factorization are deterministic, and those for equal-degree factorization are probabilistic. Indeed, it is a well-known open question to find a deterministic polynomial-time algorithm for the latter problem. In Section 9, we present a new deterministic algorithm for equal-degree factorization. Our results are best appreciated in the important special case where $q = p^k$ and p is a small, fixed prime (e.g., $p = 2$). Then our deterministic algorithm factors an arbitrary polynomial in $\mathbb{F}_q[x]$ of degree n using $O\tilde{}(n^2 + n^{3/2}k)$ operations in $\mathbb{F}_q$, and $O\tilde{}(n^2k + n^{3/2}k^2)$ bit operations. A variant extracts a single irreducible factor using just $O\tilde{}(n^2 + nk)$ operations in $\mathbb{F}_q$, and $O\tilde{}(n^2k + nk^2)$ bit operations.

Among the previously known deterministic factorization algorithms, a deterministic variant of Berlekamp's algorithm described by von zur Gathen (1987) has bit complexity $O((nk)^\omega)$, and an algorithm by Thiong ly (1989) combines the trace computations of Camion (1983) and Berlekamp (1970); its bit complexity is not stated explicitly, and appears to be $O\tilde{}(n^3k^2 + n^2k^3)$. An algorithm described by Shoup (1990) works only for $k = 1$ and has a bit complexity of $O\tilde{}(n^2)$.

2. Polynomial arithmetic

In this section, we briefly recall some facts about polynomial arithmetic, and introduce some notation.

We will let $M(n)$ denote an upper bound on the cost of polynomial multiplication. That is, we assume that we have an algorithm that multiplies two polynomials of degree n over an arbitrary ring R (commutative, containing 1) that uses $O(M(n))$ operations in R (additions, subtractions, and multiplications). We assume that this algorithm uses space for $O(n)$ elements of R.

In the analysis of some recursive algorithms, we implicitly assume that M is suitably well-behaved, namely, that M is actually a function $M: \mathbb{R}_{>0} \to \mathbb{R}_{>0}$ growing at least linearly and at most quadratically. More precisely, we insist that

$$\forall t \in \mathbb{R}_{>0}\, \forall a \in \mathbb{R}_{\geq 1} \qquad aM(t) \leq M(at) \leq a^2 M(t).$$

With the asymptotically fastest algorithms for polynomial multiplication (Schönhage & Strassen 1971, Schönhage 1977, Cantor & Kaltofen 1991), we can use $M(n) = O(n \log n \text{ loglog } n)$[1].

Let $f \in \mathbb{F}_q[x]$ have degree n, and $R = \mathbb{F}_q[x]/(f)$. For $g \in \mathbb{F}_q[x]$, we denote by $(g \bmod f)$ the image of g in R, and reserve the notation $(g \text{ rem } f)$ to denote the polynomial in $\mathbb{F}_q[x]$ obtained as the remainder on dividing g by f. For $\alpha \in R$, the *canonical representative* of α is the unique polynomial $g \in \mathbb{F}_q[x]$ of degree less than n such that $(g \bmod f) = \alpha$. At times we use the following explicit notation: for $\alpha \in R$, $\check{\alpha} \in \mathbb{F}_q[x]$ is the canonical representative of α.

To implement arithmetic in R, we use the canonical representatives, and so one addition or subtraction of elements in R takes $O(n)$ operations in $\mathbb{F}_q$. Since division with remainder of polynomials of degree at most n takes $O(M(n))$ operations (see Aho *et al.* 1974, § 8.3), multiplication of elements in R takes $O(M(n))$ operations in $\mathbb{F}_q$.

The greatest common divisor of two polynomials of degree n over $\mathbb{F}_q$ can be computed with $O(M(n) \log n)$ operations in $\mathbb{F}_q$ and space for $O(n)$ elements of $\mathbb{F}_q$ (see Aho *et al.* 1974, § 8.9, Strassen 1983). We always take the gcd of (nonzero) polynomials to be monic.

We also need an algorithm for evaluating a polynomial in $R[y]$ of degree at most n at n points in R. The "standard" fast algorithm, as in Aho *et al.* (1974, § 8.5), takes $O(M(n) \log n)$ operations in R and space for $O(n \log n)$ elements of R, which translates into $O(M(n)^2 \log n)$ operations in $\mathbb{F}_q$ and space for $O(n^2 \log n)$ elements of $\mathbb{F}_q$. In the remainder of this section, we show how to solve this problem with only $O(M(n^2) \log n)$ operations in $\mathbb{F}_q$ and space for $O(n^2)$ elements of $\mathbb{F}_q$; the reader may wish to skip this on first reading.

LEMMA 2.1. *Given a polynomial $g \in R[x]$ of degree n over an arbitrary ring R, and points $\alpha_1, \ldots, \alpha_m$ in R, with $m \leq n$, we can compute $g(\alpha_1), \ldots, g(\alpha_m)$ using*

$$O((n/m + \log m) M(m))$$

operations in R, and additional space for $O(m)$ elements of R.

[1] log denotes the natural logarithm, except that in "O"-estimates, we adopt the convention that log is defined and at least one for all positive numbers.

PROOF. We first compute the polynomial $P = (x - \alpha_1) \cdots (x - \alpha_m)$. Using a divide and conquer method, this takes $O(M(m) \log m)$ operations in R and space for $O(m)$ elements of R. Next, we compute $h = (g \text{ rem } P)$ as follows. We write $g = \sum_{0 \le i \le n/m} g_i x^{mi}$, where the degree of each $g_i \in R[x]$ is less than m. Then we successively compute higher powers of x^m, multiply in the appropriate g_i, and reduce modulo P as we go. This takes $O((n/m)M(m))$ operations in R and space for $O(m)$ elements of R.

Finally, we evaluate h at the points $\alpha_1, \ldots, \alpha_m$. This can be done using $O(M(m) \log m)$ operations in R using the algorithm in Aho *et al.* (1974, § 8.5), but that algorithm uses space for $O(m \log m)$ elements of R. We can circumvent this problem as follows. We split the points $\alpha_1, \ldots, \alpha_m$ into about $\log m$ blocks, each of size about $m/\log m$. We use the algorithm in Aho *et al.* to evaluate h at the points in the first block, and then at the points in the second block, and so on. It is easy to verify that this method uses $O(M(m) \log m)$ operations in R and space for only $O(m)$ elements of R. □

LEMMA 2.2. *Let $f \in \mathbb{F}_q[x]$ be a polynomial of degree n, and $R = \mathbb{F}_q[x]/(f)$.*

(i) Two polynomials in $R[y]$ of degree at most m can be multiplied using $O(M(mn))$ operations in $\mathbb{F}_q$.

(ii) We can evaluate a polynomial in $R[y]$ of degree at most n at m points in R, where $m \le n$, using

$$O((n/m + \log m)M(mn))$$

operations in $\mathbb{F}_q$ and space for $O(nm)$ elements in $\mathbb{F}_q$.

PROOF. (i) Let $g, h \in R[y]$ be the polynomials to be multiplied. We can view the coefficients of g and h as elements of $\mathbb{F}_q[x]$, compute the product in $\mathbb{F}_q[x, y]$, and reduce each coefficient in the product modulo f. To compute the product polynomial in $\mathbb{F}_q[x, y]$, we can perform the well-known Kronecker substitution $y \mapsto x^{2n-1}$, reducing our problem to that of multiplying two univariate polynomials in $\mathbb{F}_q[x]$ of degree at most $(2n - 1)m$.

Using this method, the cost of computing the product (as a polynomial in $\mathbb{F}_q[x, y]$) is $O(M(mn))$ operations in $\mathbb{F}_q$, and the cost of reducing the coefficients in the product polynomial modulo f is $O(mM(n))$ operations in $\mathbb{F}_q$, which is $O(M(mn))$.

Assertion (ii) follows directly from (i) and the proof of Lemma 2.1. □

Remark. The problem of multiplying polynomials in $R[y]$ of degree at most m is in fact asymptotically equivalent to the problem of multiplying polynomials in $\mathbb{F}_q[x]$ of degree at most mn, in the sense that an algorithm for the former problem can be used to solve the latter problem using the same number of operations in $\mathbb{F}_q$ (up to a constant multiplicative factor).

3. A factoring algorithm

The purpose of this section is to describe and analyze an algorithm that will factor a polynomial $f \in \mathbb{F}_q[x]$ of degree n with $O\tilde{~}(n^2 + n \log q)$ operations in $\mathbb{F}_q$.

The squarefree factorization of f is of the form $f = f_1 f_2^2 f_3^3 \cdots f_n^n$, where $f_1, \ldots, f_n$ are monic, squarefree, and pairwise relatively prime. Thus, f_i is the product of those monic irreducible polynomials in $\mathbb{F}_q[x]$ that divide f exactly to the power i. Yun's (1976) algorithm computes this squarefree factorization with $O(M(n) \log n + n \log q)$ operations in $\mathbb{F}_q$ and space for $O(n)$ elements of $\mathbb{F}_q$ (see Knuth 1981, Exercise 4.6.2–36). So we may now assume that f is squarefree.

In most of the paper, we will use the following notation.

$$\begin{gathered} q \text{ is a prime power, } f \in \mathbb{F}_q[x] \text{ is a squarefree monic} \\ \text{polynomial of degree } n,\ R = \mathbb{F}_q[x]/(f), \text{ and } \xi = (x \bmod f) \in R. \end{gathered} \tag{3.1}$$

Our basic algorithms for distinct-degree and equal-degree factorization use as a subroutine an algorithm for computing iterates of the Frobenius map on R, which is based on the following observation. If for some $m \leq n$ we have computed $\xi, \xi^q, \ldots, \xi^{q^m}$ and $g \in \mathbb{F}_q[x]$ is the canonical representative of ξ^{q^m}, then we can calculate $\xi^{q^{m+1}}, \ldots, \xi^{q^{2m}}$ by evaluating the polynomial g at the points $\xi^q, \ldots, \xi^{q^m}$, since $g(\xi^{q^i}) = \xi^{q^{m+i}}$. This "doubling step" can be done using a fast multi-point evaluation algorithm with $O\tilde{~}(n)$ operations in R. Thus, if we first compute ξ^q using repeated squaring—which takes $O(\log q)$ operations in R—and then perform the above "doubling step" $O(\log n)$ times, we can compute all of the elements $\xi, \xi^q, \ldots, \xi^{q^n}$ with $O\tilde{~}(n + \log q)$ operations in R, and hence $O\tilde{~}(n^2 + n \log q)$ operations in $\mathbb{F}_q$.

Algorithm 3.1. *Iterated Frobenius.*
Input: *f, α, β, and m, where, in the notation (3.1), α and β are elements of R with $\beta = \xi^t$ for some power t of q, and m is a positive integer with $m \leq n$.*
Output: *The elements $\alpha, \alpha^t, \ldots, \alpha^{t^m}$ in R.*

1. *Set* $\gamma_0 = \xi \in R$, *and* $\gamma_1 = \beta$. *Let* $\ell = \lceil \log_2 m \rceil$.

2. *For* $i = 1, \ldots, \ell$ *do the following. [After stage* i, $\gamma_j = \xi^{t^j}$ *has been computed for* $0 \leq j \leq 2^i$.*]*

   Compute $\gamma_{2^{i-1}+j} = \check{\gamma}_{2^{i-1}}(\gamma_j)$ *for* $j = 1, \ldots, 2^{i-1}$ *using a fast multi-point evaluation algorithm.*

3. *Compute* $\gamma'_j = \check{\alpha}(\gamma_j)$ *for* $j = 0, \ldots, m$ *using a fast multi-point evaluation algorithm.*

4. *Return* $\gamma'_0, \gamma'_1, \ldots, \gamma'_m$.

THEOREM 3.2. *Algorithm 3.1 works correctly as specified, and uses*

$$O\left(\frac{n}{m} M(mn) \log m\right)$$

operations in $\mathbb{F}_q$ *and space for* $O(n\,m)$ *elements of* $\mathbb{F}_q$. *In particular, the running time can be bounded by*

$$O(n^2 \log n \ \operatorname{loglog} n \ \log m)$$

operations in $\mathbb{F}_q$.

PROOF. For the correctness, we first prove by induction on i the assertion made in step 2 of the algorithm, namely that $\gamma_j = \xi^{t^j}$ for $0 \leq j \leq 2^i$. We have for $1 \leq j \leq 2^{i-1}$

$$\begin{aligned} \gamma_{2^{i-1}+j} &= \check{\gamma}_{2^{i-1}}(\gamma_j) = \check{\gamma}_{2^{i-1}}(\xi^{t^j}) = (\check{\gamma}_{2^{i-1}}(\xi))^{t^j} \\ &= (\gamma_{2^{i-1}})^{t^j} = (\xi^{t^{2^{i-1}}})^{t^j} = \xi^{t^{2^{i-1}+j}}, \end{aligned}$$

and thus for $0 \leq j \leq m$

$$\gamma'_j = \check{\alpha}(\gamma_j) = \check{\alpha}(\xi^{t^j}) = (\check{\alpha}(\xi))^{t^j} = \alpha^{t^j}.$$

By Lemma 2.2(ii), stage i of step 2 of takes

$$O\left((n/2^i + i) M(2^i n)\right)$$

operations in $\mathbb{F}_q$; moreover, the cost of step 3 is asymptotically bounded by the cost of the last stage of step 2. Summing over $1 \leq i \leq \ell$ yields the running time bound in the statement of the theorem. The assertions about space are clear. □

For distinct-degree factorization, we use a variant of an algorithm that already appears in Arwin (1918). As an aside, we remark that Arwin's fascinating paper foreshadows many other methods used in modern polynomial factorization, including trace computation (McEliece 1969, Ben-Or 1981, Camion 1983), reduction to root-finding via resultants (Berlekamp 1970), and distinguishing roots by their order (Moenck 1977, von zur Gathen 1987, Mignotte & Schnorr 1988, Menezes et al. 1992). Arwin's algorithm is based on the following fact (see Lidl & Niederreiter 1983, Theorem 3.20).

FACT 3.3. *For $d \geq 1$, the polynomial $x^{q^d} - x \in \mathbb{F}_q[x]$ is the product of the monic irreducible polynomials in $\mathbb{F}_q[x]$ whose degree divides d.*

ALGORITHM 3.4. *Distinct-degree factorization.*
Input: *A squarefree monic polynomial f over $\mathbb{F}_q$ of degree n.*
Output: *The set of all pairs (g, d) such that g is the product of all monic irreducible factors of f of degree d with $g \neq 1$.*

1. *For ξ as in (3.1), compute $\xi^q \in R$ using a repeated squaring algorithm.*
2. *Using Algorithm 3.1 with arguments (f, ξ, ξ^q, n), compute $\alpha_i = \xi^{q^i}$ for $i = 1, \ldots, n$.*
3. *Put $S = \emptyset$, put $f^* = f$, and for $i = 1, \ldots, n$, do the following.*
   - *(i) Compute $g = \gcd(f^*, \check{\alpha}_i - x)$.*
   - *(ii) If $g \neq 1$, add (g, i) to S.*
   - *(iii) Replace f^* with f^*/g.*
4. *Return S.*

THEOREM 3.5. *Algorithm 3.4 works correctly as specified, and uses*

$$O(M(n^2) \log n + M(n) \log q)$$

operations in $\mathbb{F}_q$ and space for $O(n^2)$ elements of $\mathbb{F}_q$. In particular, the running time can be bounded by

$$O((n^2 \log n + n \log q) \cdot \log n \ \operatorname{loglog} n)$$

operations in $\mathbb{F}_q$.

PROOF. Note that $\check{\alpha}_i - x \equiv x^{q^i} - x \bmod f$, so that the correctness of the algorithm follows from Fact 3.3. The time and space bounds are easy to verify. □

ALGORITHM 3.6. *Equal-degree factorization.*
Input: *A squarefree polynomial f over $\mathbb{F}_q$ of degree n and a positive integer d, where f is the product of r irreducible factors, each of degree d (so that $n = rd$).*
Output: *The set of monic irreducible factors of f.*

1. *If* $\deg f = d$, *then return* $\{f\}$; *if* $\deg f = 0$, *then return* $\emptyset$.
2. *For ξ as in (3.1), compute ξ^q using repeated squaring.*
3. *Pick $\alpha \in R$ at random and compute*

$$\beta = \sum_{0 \le i < d} \alpha^{q^i},$$

using Algorithm 3.1 with arguments $(f, \alpha, \xi^q, d-1)$ to get $\alpha, \alpha^q, \ldots, \alpha^{q^{d-1}}$, and then summing these elements.

4. *If q is odd, then do the following:*

   (i) *Compute $\gamma = \beta^{(q-1)/2}$ using repeated squaring.*
   (ii) *Compute $h_1 = \gcd(\check{\gamma}, f)$, $h_2 = \gcd(\check{\gamma} - 1, f)$, and $h_3 = f/(h_1 h_2)$.*
   (iii) *Recursively factor h_1, h_2, and h_3, and return the union of these three sets of factors.*

   Else, if $q = 2^k$, do the following:

   (i) *Compute $\gamma = \sum_{0 \le i < k} \beta^{2^i}$.*
   (ii) *Compute $h_1 = \gcd(\check{\gamma}, f)$ and $h_2 = f/h_1$.*
   (iii) *Recursively factor h_1 and h_2, and return the union of these two sets of factors.*

THEOREM 3.7. *Algorithm 3.6 works correctly as specified, and uses an expected number of*

$$O(M(nd) r \log d + M(n) \log r \, \log q)$$

operations in $\mathbb{F}_q$, an expected number of $O(n \log r)$ random elements of $\mathbb{F}_q$, and space for $O(nd)$ elements of $\mathbb{F}_q$. In particular, the expected running time can be bounded by

$$O((n^2 \log d + n \log r \ \log q) \cdot \log n \ \operatorname{loglog} n)$$

operations in $\mathbb{F}_q$.

PROOF. *Analyzing a single recursive invocation.* We consider first a single invocation of the body of this recursive algorithm. Step 2 takes $O(M(n) \log q)$ operations in $\mathbb{F}_q$, step 3 takes $O(rM(nd) \log d)$ operations in $\mathbb{F}_q$, and step 4 takes $O(M(n) \log n + M(n) \log q)$ operations in $\mathbb{F}_q$.

Let $f_1, \ldots, f_r \in \mathbb{F}_q[x]$ be the irreducible factors of f. Then by the Chinese Remainder Theorem, we have an isomorphism from R onto $\oplus_{i=1}^r \mathbb{F}_q[x]/(f_i) \cong \oplus_{i=1}^r \mathbb{F}_{q^d}$ that maps $\mathbb{F}_q$ onto the diagonal. For $\alpha \in R$, suppose that $(\alpha_1, \cdots, \alpha_r)$, with each $\alpha_i \in \mathbb{F}_{q^d}$, is the image of α under this isomorphism.

For α randomly chosen in step 3, $\alpha_1, \ldots, \alpha_r$ are independently and uniformly distributed random elements in $\mathbb{F}_{q^d}$. If $T \colon \mathbb{F}_{q^d} \to \mathbb{F}_q$ denotes the trace, then $\beta_i = T(\alpha_i)$ for all i. Since T is $\mathbb{F}_q$-linear and surjective, $\beta_1, \ldots, \beta_r$ are independently and uniformly distributed random elements in $\mathbb{F}_q$.

In step 4, suppose first that q is odd. Each γ_i is 0 with probability $1/q$, and otherwise is ± 1 with equal probability. The polynomials h_1, h_2, h_3 are the products of those f_i with γ_i equal to 0, 1, -1, respectively. If q is even, each component of γ is 0 or 1 with equal probability. In this case, h_1 and h_2 are the products of those f_i with γ_i equal to 0 and 1, respectively.

Analyzing the entire computation. One easily checks that the recursion tree has expected depth $O(\log r)$, since any pair of factors is "split" in one invocation with probability at least 1/2. From this, one obtains a bound of

$$O(\log r \cdot rM(nd) \log d + M(n) \log r \ \log q)$$

on the expected number of operations in $\mathbb{F}_q$. This proves the theorem, except for the additional factor of $\log r$ in the first summand.

We now give a more detailed analysis of the algorithm that eliminates this extra factor of $\log r$. We model the probabilistic behavior of the algorithm by a certain type of "balls and bins" game; we refer to the following section for a description of this game.

Suppose that q is odd. We calculate first the expected number of operations in $\mathbb{F}_q$ spent performing step 3 in *all* recursive invocations of the algorithm.

This corresponds to a balls and bins game in the following way. The "balls" correspond to the irreducible factors of f. The "bins" correspond to the polynomials h_1, h_2, and h_3. A "ball landing in bin i" corresponds to an irreducible factor of f dividing h_i. The probabilities associated with the three bins are $p_1 = 1/q$, $p_2 = (q-1)/2q$, and $p_3 = (q-1)/2q$.

The cost function g of the game is chosen so that $g(r)$ is an upper bound on the number of operations performed in step 3 in a single recursive invocation of the algorithm. We can take

$$g(r) = c \cdot rM(rd^2)(\log d + 1),$$

for an appropraitely chosen constant c. Clearly, g satisfies the strong superlinearity condition of Lemma 4.1(ii), and we then find (with $\ell = 3$, $w = 1/2$, and $\epsilon = 1$) that the expected number of operations spent performing step 3 in all recursive invocations is $O(rM(nd)\log d)$.

By similar reasoning, we can bound the expected number of operations spent in steps 2 and 4 in all recursive invocations by $O(M(n)\log n\ \log r + M(n)\log r\ \log q)$. Note that we incur the extra factor of $\log r$ because the associated cost function will not necessarily satisfy the hypothesis of Lemma 4.1(ii), and so only (i) applies.

If q is even, we can argue similarly, but this time the behavior of the algorithm is modeled by a balls and bins game with two bins, each with associated probability $1/2$. □

We summarize the results of this section in the following theorem, which is the main result of this paper.

THEOREM 3.8. *We have a probabilistic algorithm that factors a polynomial of degree n over $\mathbb{F}_q$ using an expected number of*

$$O((n^2 + n\log q) \cdot (\log n)^2 \operatorname{loglog} n)$$

operations in $\mathbb{F}_q$.

4. A game of balls and bins

In the proof of Theorem 3.7, it is convenient to model the probabilistic behavior of Algorithm 3.6 by a balls and bins game of the following type. We have a fixed number ℓ of bins, and when a ball is tossed, the probability that it lands in bin i is p_i for $1 \leq i \leq \ell$. A ball must land in some bin, i.e., $\sum_{1\leq i\leq \ell} p_i = 1$, and $p_i \leq w < 1$ for all i. Associated with the game is a "cost function" $g : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$.

The game is played with $r \geq 2$ balls as follows. First, the player pays $g(r)$ units. Second, the player throws the balls into the bins. Third, for each bin that contains at least two balls, the player recursively plays the game with the balls in that bin.

Let $G(r)$ be the total number of units that the player pays during the game, including the units paid in all recursive games. We want to estimate the expected value $E[G(r)]$ of $G(r)$.

LEMMA 4.1. *(i) If g grows at least linearly, i.e.,*

$$\forall t \in \mathbb{R}_{>0}\, \forall a \in \mathbb{R}_{\geq 1} \quad g(at) \geq a g(t),$$

then

$$\forall r \geq 2 \quad E[G(r)] \leq c_1 g(r) \log r,$$

where c_1 is a constant depending only on w.

(ii) If g grows at a strongly super-linear rate, i.e., for some constant $\epsilon \in \mathbb{R}_{>0}$

$$\forall t \in \mathbb{R}_{>0}\, \forall a \in \mathbb{R}_{\geq 1} \quad g(at) \geq a^{1+\epsilon} g(t),$$

then

$$\forall r \geq 2 \quad E[G(r)] \leq c_2 g(r),$$

where c_2 is a constant depending only on w, ℓ, and ϵ.

PROOF. Corresponding to any game is a tree representing all of the recursive games played, where each node in the tree is labeled with the number of units payed for that particular game. The quantity $G(r)$ is just the sum of all of the values labeling the nodes of the tree. Let D be the number of levels in the game tree.

We first bound the expected value $E[D]$ of D. For any pair of balls, the probability that they land in the same bin in any one game is at most w. Thus, the probability that this pair of balls has not been split after level number t in the game tree is at most w^t. Since the number of such pairs is less than r^2, we have $\Pr[D \geq t] < r^2 w^{t-1}$. Then one calculates that

$$\begin{aligned} E[D] &= \sum_{t \geq 1} \Pr[D \geq t] \\ &\leq \frac{2 \log r}{\log(1/w)} + 1 + \sum_{t > 2\log r/\log(1/w)+1} r^2 w^{t-1} \\ &\leq \frac{2 \log r}{\log(1/w)} + 1 + 1/(1-w). \end{aligned}$$

Our assumption that g grows at least linearly implies that the values labeling the nodes on any one level of the tree sum to no more than $g(r)$. Thus, $E[G(r)] \leq g(r) \cdot E[D]$, and (i) follows with

$$c_1 = \frac{2}{\log(1/w)} + \frac{1 + 1/(1-w)}{\log 2}.$$

For (ii), we first note that $\sum_{1 \leq i \leq \ell} p_i^{1+\epsilon} \leq w^\epsilon < 1$. Let $\gamma = (1 + w^\epsilon)/2$. Clearly, we can choose $\delta > 0$—depending only on w, ϵ and ℓ—in such a way that $\sum_{1 \leq i \leq \ell} (p_i + \delta)^{1+\epsilon} < \gamma$.

Call a toss of the balls "bad" if more than $p_i r + \delta r$ balls land in bin i for some $i \leq \ell$. By using well-known bounds on the tail of the binomial distribution (see Cormen *et al.* 1989, Corollary 6.7) one easily calculates that the probability of a bad toss is no more than $\ell\, e^{-\delta^2 r}$.

Let c_1 be as above, $A = \max\{c_1 \log r \ell e^{-\delta^2 r} : r \geq 2\}$, and $c_2 = (1+A)/(1-\gamma)$. We prove by induction on r that $E[G(r)] \leq c_2 g(r)$ for all $r \geq 2$. One easily verifies the base case $r = 2$. Then, for $r > 2$, we have

$$\begin{aligned} E[G(r)] &= E[G(r) \mid \text{bad toss}] \cdot \Pr[\text{bad toss}] \\ &\quad + E[G(r) \mid \text{good toss}] \cdot \Pr[\text{good toss}] \\ &\leq g(r) + c_1 g(r) \log r \cdot \ell e^{-\delta^2 r} + \sum_{1 \leq i \leq \ell} c_2 g(p_i r + \delta r) \\ &\leq g(r)(1 + A) + c_2 g(r) \sum_{1 \leq i \leq \ell} (p_i + \delta)^{1+\epsilon} \\ &\leq g(r)(1 + A + c_2 \gamma) \;=\; c_2 g(r). \qquad \Box \end{aligned}$$

5. Equal-degree factorization

In this section, we consider again the equal-degree factorization problem. We describe a method that is more efficient than Algorithm 3.6, in terms of both time and space. One variant of this algorithm uses $O\tilde{}(n^2 + n \log q)$ operations in $\mathbb{F}_q$ and space for $O(n)$ elements of $\mathbb{F}_q$. Another variant uses $O\tilde{}(n^{(\omega+1)/2} + n \log q)$ operations in $\mathbb{F}_q$ and space for $O(n \log n)$ elements of $\mathbb{F}_q$.

We now denote the cost of matrix multiplication by $MM(n)$ rather than n^ω; that is, we assume that we have an algorithm that multiplies two $n \times n$ matrices over an arbitrary ring R using $O(MM(n))$ operations in R. We assume that the algorithm uses space for $O(n^2)$ elements of R, and that MM is actually a function $MM\colon \mathbb{R}_{>0} \to \mathbb{R}_{>0}$ such that

$$\forall t \in \mathbb{R}_{>0}\ \forall a \in \mathbb{R}_{\geq 1} \qquad a^2 MM(t) \leq MM(at) \leq a^3 MM(t).$$

We can choose $MM(n) = n^{2.376}$ (Coppersmith & Winograd 1990).

A basic problem for the algorithms in this section is *modular composition*: computing $g(h)$ rem f, where f, g, and h are polynomials over $\mathbb{F}_q$. We summarize what is known about the complexity of this problem.

FACT 5.1. *Let f, g, and h be polynomials over $\mathbb{F}_q$ of degree at most n. Then we can compute $g(h)$ rem f in the following time and space bounds (time is measured in operations in $\mathbb{F}_q$, and space is measured in storage for elements of $\mathbb{F}_q$):*

(i) time $O(nM(n))$ and space $O(n)$,

(ii) time $O(n^{1/2}(M(n) + MM(n^{1/2})))$ and space $O(n^{3/2})$,

(iii) time $O((MM(n^{1/2})\, n^{1/2} \log n + n^{3/2} \log n\, M(\ell)\, \ell^{-1} + M(n)(\log n)^3)$ and space $O(n \log n)$, where $\ell = \lceil \log n / \log q \rceil$.

PROOF. (i) is proved by viewing the problem as that of evaluating a polynomial of degree n at a single point in the ring $R = \mathbb{F}_q[X]/(f)$. Using Horner's rule, this takes $O(n)$ operations in R and space for $O(1)$ elements of R. Algorithm 2.1 in Brent & Kung (1978) solves this problem for the special case where $f = x^n$ within the time and space bounds stated in (ii). In fact, this algorithm works as stated for arbitrary f. (iii) is proved in Shoup & Smolensky (1992). $\square$

The complexity of modular composition is not well understood. In the special case where $f = x^n$, Algorithm 2.2 in Brent & Kung (1978) solves this problem using $O\tilde{}(n^{3/2})$ operations. Moreover, no super-linear lower bounds are known for this problem.

In what follows we shall use $C_T(n)$ and $C_S(n)$ to denote the time and space bounds for a modular composition algorithm. We assume that C_T is a function $C_T \colon \mathbb{R}_{>0} \to \mathbb{R}_{>0}$ such that

$$\forall t \in \mathbb{R}_{>0}\ \forall a \in \mathbb{R}_{\geq 1} \qquad aC_T(t) \leq C_T(at) \leq a^3 C_T(t).$$

We also assume that C_S is nondecreasing, and that $C_S(n) \geq n$.

The bottleneck in Algorithm 3.6 is the application of the iterated Frobenius algorithm in step 3 to compute $\alpha, \alpha^q, \ldots, \alpha^{q^{d-1}}$. All we really need to compute is the sum of these elements, i.e., the trace map $\sum_{0 \leq i < d} \alpha^{q^i}$. We now present a different algorithm for computing trace maps, based on the following observation. Recall (3.1) and the notation $\check{\alpha}$ for the canonical representative of an element α of R. For $m \geq 1$, let $\beta_m = \sum_{0 \leq i < m} \alpha^{q^i}$ and $\gamma_m = \xi^{q^m}$. Then $\beta_{2m} = \beta_m + \check{\beta}_m(\gamma_m)$

and $\gamma_{2m} = \check{\gamma}_m(\gamma_m)$. We can therefore obtain β_{2m} and γ_{2m} from β_m and γ_m at a cost of two modular compositions (plus one addition in R). Thus, if d is a power of 2, we can easily compute the trace map $\beta_d = \sum_{0\le i<d} \alpha^{q^i}$ as follows: first, compute ξ^q using repeated squaring, and then repeat the above "doubling step" $\log_2 d$ times. The total cost of this method is $O\tilde{~}(n \log q)$ operations in $\mathbb{F}_q$ plus $O(\log d)$ modular compositions. The algorithm is easily adapted to deal with arbitrary d.

ALGORITHM 5.2. *Computing trace maps.*
Input: *f, α, β, and m, where, in the notation (3.1), α and β are elements of R with $\beta = \xi^t$ for some power t of q, and m is a positive integer.*
Output: *The elements α^{t^m} and $\sum_{0\le j\le m} \alpha^{t^j}$ in R.*

1. *Let $m = \sum_{0\le i\le \ell} b_i 2^i$ be the binary representation of m, with $\ell = \lfloor \log_2 m \rfloor$ and $b_0, \ldots, b_\ell \in \{0, 1\}$.*

2. *(stage 0):*

$$\tau \leftarrow \check{\alpha}(\beta), \qquad \tau' \leftarrow \begin{cases} \alpha & \text{if } b_0 = 0, \\ \alpha + \tau & \text{if } b_0 = 1, \end{cases}$$

$$\mu \leftarrow \beta, \qquad \mu' \leftarrow \begin{cases} \xi & \text{if } b_0 = 0, \\ \beta & \text{if } b_0 = 1. \end{cases}$$

3. *For $i = 1, \ldots, \ell$, do the following (stage i):*

$$\tau \leftarrow \tau + \check{\tau}(\mu), \qquad \tau' \leftarrow \begin{cases} \tau' & \text{if } b_i = 0, \\ \tau' + \check{\tau}(\mu') & \text{if } b_i = 1, \end{cases}$$

$$\mu \leftarrow \check{\mu}(\mu), \qquad \mu' \leftarrow \begin{cases} \mu' & \text{if } b_i = 0, \\ \check{\mu}(\mu') & \text{if } b_i = 1. \end{cases}$$

4. *Return $\check{\alpha}(\mu')$ and τ'.*

LEMMA 5.3. *Algorithm 5.2 works correctly as specified, and uses $O(C_T(n) \cdot \log m)$ operations in $\mathbb{F}_q$ and space for $O(C_S(n))$ elements of $\mathbb{F}_q$.*

PROOF. The time and space bounds are clear, as the algorithm performs $O(\log m)$ modular compositions and additions of polynomials of degree at most n.

To prove correctness, let $m_i = \sum_{0\le j\le i} b_j 2^j$ for $0 \le i \le \ell$, and let $\tau_i, \mu_i, \tau_i', \mu_i'$ denote the values of the variables τ, μ, τ', μ' at the end of stage i. We show by induction on i that the following identities hold:

$$\begin{aligned} \tau_i &= \sum_{1\le j\le 2^i} \alpha^{t^j}, & \tau_i' &= \sum_{0\le j\le m_i} \alpha^{t^j}, \\ \mu_i &= \xi^{t^{2^i}}, & \mu_i' &= \xi^{t^{m_i}}. \end{aligned}$$

The base case $i = 0$ is easily verified. Now assume that the identities hold after stage i; we show that they also hold after stage $i+1$. We have

$$\begin{aligned} \tau_{i+1} &= \tau_i + \check{\tau}_i(\mu_i) = \tau_i + \check{\tau}_i(\xi^{t^{2^i}}) = \tau_i + \tau_i^{t^{2^i}} \\ &= \sum_{1\le j\le 2^i} \alpha^{t^j} + \sum_{2^i < j \le 2^{i+1}} \alpha^{t^j} = \sum_{1\le j\le 2^{i+1}} \alpha^{t^j}, \\ \mu_{i+1} &= \check{\mu}_i(\mu_i) = \check{\mu}_i(\xi^{t^{2^i}}) = \mu_i^{t^{2^i}} = \xi^{t^{2^{i+1}}}. \end{aligned}$$

If $b_{i+1} = 0$, it is clear that τ_{i+1}' and μ_{i+1}' will have the correct values, so assume that $b_{i+1} = 1$. Then we have

$$\begin{aligned} \tau_{i+1}' &= \tau_i' + \check{\tau}_{i+1}(\mu_i') = \tau_i' + \check{\tau}_{i+1}(\xi^{t^{m_i}}) = \tau_i' + \tau_{i+1}^{t^{m_i}} \\ &= \tau_i' + \Big(\sum_{1\le j\le 2^{i+1}} \alpha^{t^j}\Big)^{t^{m_i}} = \sum_{0\le j\le m_i} \alpha^{t^j} + \sum_{m_i < j \le m_i + 2^{i+1}} \alpha^{t^j} = \sum_{0\le j\le m_{i+1}} \alpha^{t^j}, \\ \mu_{i+1}' &= \check{\mu}_{i+1}(\mu_i') = \check{\mu}_{i+1}(\xi^{t^{m_i}}) = \mu_{i+1}^{t^{m_i}} = \xi^{t^{m_i + 2^{i+1}}} = \xi^{t^{m_{i+1}}}. \qquad \square \end{aligned}$$

We now modify Algorithm 3.6 by replacing step 3 with the following.

3′. Pick $\alpha \in R$ at random, and compute $\beta = \sum_{0\le i<d} \alpha^{q^i}$ using Algorithm 5.2 with arguments $(f, \alpha, \xi^q, d-1)$.

THEOREM 5.4. *With the above modification, Algorithm 3.6 uses an expected number of*

$$O(C_T(n)\log d\, \log r + M(n)\log n\, \log r + M(n)\log r\, \log q)$$

operations in $\mathbb{F}_q$ and space for $O(C_S(n))$ elements of $\mathbb{F}_q$. Moreover, if C_T is one of the bounds in Fact 5.1, then this modified version of Algorithm 3.6 uses an expected number of

$$O(C_T(n)\log d + M(n)\log r\, \log q)$$

operations in $\mathbb{F}_q$. In particular, we have the following time and space bounds (time is measured in operations in $\mathbb{F}_q$ and space in storage for elements of $\mathbb{F}_q$):

(i) time $O(n^2 \log n \ \mathrm{loglog}\, n \ \log d + n \log n \ \mathrm{loglog}\, n \ \log r \ \log q)$ *and space* $O(n)$*;*

(ii) time $O(n^{1.7} + n \log n \ \mathrm{loglog}\, n \ \log r \ \log q)$ *and space* $O(n \log n)$*.*

PROOF. The proof is essentially the same as that of Theorem 3.7, and we omit the details. The exponent in (ii) comes from $(2.376 + 1)/2 = 1.688 < 1.7$. □

6. Distinct-degree factorization

In this section, we describe another algorithm for distinct-degree factorization that uses $O\tilde{}(n^2 + n \log q)$ operations in $\mathbb{F}_q$, like Algorithm 3.4, but space for only $O(n^{3/2})$ elements of $\mathbb{F}_q$.

LEMMA 6.1. *Let* $m \le n$, $g_1, \ldots, g_m \in \mathbb{F}_q[x, y]$ *with* $\deg_x g_i \le 1$ *and* $\deg_y g_i < n$ *for all* $i \le m$*, and* $h \in \mathbb{F}_q[y]$ *be monic with* $\deg h \le n$*. Then*

$$P = \Big(\prod_{1 \le i \le m} g_i \Big) \ \mathrm{rem}\ h \in \mathbb{F}_q[x, y]$$

can be computed with

$$O(M(mn) \log m)$$

operations in $\mathbb{F}_q$ *and space for* $O(mn)$ *elements of* $\mathbb{F}_q$*.*

PROOF. The polynomial P is obtained by taking the remainder modulo h of each coefficient in $\prod_{1 \le i \le m} g_i$ at a power of x.

To compute it, we use a binary tree with $\lceil \log_2 m \rceil$ levels and $g_1, \ldots, g_m$ at the leaves, i.e., level zero. At each node, we multiply the two children and take the remainder modulo h of each coefficient of a power of x. The degree in x at level $j - 1$ is at most 2^{j-1}, and that in y is less than n. The claims follow from Lemma 2.2(i), with the rôles of x and y interchanged and $R = \mathbb{F}_q[x]/(h)$. □

We now describe a subroutine used by our distinct-degree factorization algorithm.

ALGORITHM 6.2. *Range decomposition.*
Input: f*,* a*,* b*,* α*,* β*, where* a *and* b *are positive integers with* $a \le b$*, and* $\alpha = \xi^{q^a}$ *and* $\beta = \xi^q$*, in the notation (3.1); it is assumed that the degrees of all irreducible factors of* f *lie in the range* $a, \ldots, b$*.*
Output: *The set of all pairs* (g, d) *such that* g *is the product of all monic irreducible factors of* f *of degree* d *with* $g \ne 1$*.*

1. *If $a > n$, return $\emptyset$; if $b > n$, replace b with n.*
2. *Use Algorithm 3.1 with arguments $(f, \alpha, \beta, b-a)$ to compute $\alpha_i = \xi^{q^i}$ for $i = a, \ldots, b$.*
3. *Put $S = \emptyset$, $f^* = f$, and do the following for $i = a, \ldots, b$.*
   (i) *Compute $g = \gcd(f^*, \check{\alpha}_i - x)$.*
   (ii) *If $g \neq 1$, add (g, i) to S.*
   (iii) *Replace f^* with f^*/g.*
4. *Return S.*

LEMMA 6.3. *Algorithm 6.2 works correctly as specified, and uses*

$$O\left(\frac{n}{m} M(mn) \log m\right)$$

operations in $\mathbb{F}_q$ and space for $O(nm)$ elements of $\mathbb{F}_q$, where $m = \min\{n, b-a+1\}$. In particular, the running time can be bounded by

$$O(n^2 \log n \, \operatorname{loglog} n \, \log m)$$

operations in $\mathbb{F}_q$.

PROOF. The correctness follows from Fact 3.3. The time and space bounds can be easily checked. □

Our distinct-degree factorization algorithm works by dividing the interval $1, \ldots, n$ into roughly $\sqrt{n}$ intervals each of size about $\sqrt{n}$. For each interval, it finds the product of the irreducible factors whose degree lies in the that interval, and then applies Algorithm 6.2 to get the complete distinct-degree factorization.

ALGORITHM 6.4. *Distinct-degree factorization.*
Input: *A squarefree polynomial $f \in \mathbb{F}_q[x]$ of degree n.*
Output: *The set of all pairs (g, d) such that g is the product of all monic irreducible factors of f of degree d with $g \neq 1$.*

1. *In the notation (3.1), compute ξ^q using repeated squaring.*
2. *Let $m = \lfloor n^{1/2} \rfloor$, and $m' = \lceil n/m \rceil$.*

3. *Apply Algorithm 3.1 with arguments* (f, ξ, ξ^q, m) *to obtain* $\mu_j = \xi^{q^j}$ *for* $j = 0, \ldots, m$.

4. *Apply Algorithm 3.1 with arguments* $(f, \xi^q, \mu_m, m' - 1)$ *to obtain* $\nu_i = \xi^{q^{im+1}}$ for $i = 0, \ldots, m' - 1$.

5. *Compute the polynomial*

$$g(y) = \prod_{0 \leq j < m} (\check{\mu}_j(y) - \xi) \text{ rem } f(y) \in R[y]$$

using the method described in Lemma 6.1. [Then $g(y) \equiv \prod_{0 \leq j < m} (y^{q^j} - \xi) \bmod f(y)$.*]*

6. *Compute* $\tau_i = g(\nu_i) \in R$ *for* $i = 0, \ldots, m' - 1$ *using a fast multi-point evaluation algorithm. [Then* $\tau_i = \prod_{0 \leq j < m} (\xi^{q^{im+1+j}} - \xi)$.*]*

7. *Put* $S = \emptyset$, $f^* = f$, *and for* $i = 0, \ldots, m' - 1$, *do the following:*

   (i) Compute $h = \gcd(\check{\tau}_i, f^*)$.

   (ii) Replace f^* *with* f^*/h.

   (iii) If $h \neq 1$, *compute the distinct-degree factorization of* h *by invoking Algorithm 6.2 with arguments* $(h, im + 1, im + m, \nu_i, \mu_1)$ *and add the result to* S.

8. *Return* S.

THEOREM 6.5. *Algorithm 6.4 works correctly as specified, and uses*

$$O(M(n^{3/2})n^{1/2} \log n + M(n) \log q)$$

operations in $\mathbb{F}_q$ *and space for* $O(n^{3/2})$ *elements of* $\mathbb{F}_q$. *In particular, the running time can be bounded by*

$$O((n^2 \log n + n \log q) \cdot \log n \ \operatorname{loglog} n)$$

operations in $\mathbb{F}_q$.

PROOF. We first prove correctness. The only subtle point is that the assertion made in step 6 holds because we are evaluating the polynomial $g(y)$ at points

ν_i that are zeros of $f(y)$. Thus, $g(\nu_i) = \prod_{0 \le j < m}(\nu_i^{q^j} - \xi)$. Consider the ith iteration of the loop in step 7. We have

$$\check{\tau}_i \equiv \prod_{0 \le j < m} (x^{q^{im+1+j}} - x) \pmod{f}.$$

It follows from Fact 3.3 that the polynomial h computed in the ith iteration is the product of those irreducible factors of f whose degree lies in the range $im+1, \ldots, im+m$. The invocation of Algorithm 6.2 then correctly computes the distinct-degree decomposition of h.

We now estimate the running time, measured in terms of operations in $\mathbb{F}_q$.

- Step 1 takes time $O(M(n) \log q)$.
- Steps 3 and 4 take time $O(M(n^{3/2})n^{1/2} \log n)$, by Theorem 3.2.
- Step 5 takes time $O(M(n^{3/2}) \log n)$, by Lemma 6.1.
- Step 6 takes time $O(M(n^{3/2})n^{1/2})$, by Lemma 2.2(ii).
- The total time spent in steps 7(a) and 7(b) is $O(n^{1/2}M(n) \log n)$.

  Implicit in step 7(c) is the computation at the ith iteration of $\check{\mu}_1$ rem h and $\check{\nu}_i$ rem h. The total time spent doing this is $O(n^{1/2}M(n))$. Suppose that Algorithm 6.2 is invoked with polynomials of degree n_i for $0 \le i < m'$; then $\sum_{0 \le i < m'} n_i = n$. The ith invocation takes time $O(M(n^{1/2}n_i)\, n^{1/2} \log n)$. The sum of this quantity over all values of i is $O(M(n^{3/2})\, n^{1/2} \log n)$, which also bounds the total time for step 7.

Thus the running time is bounded by $O(M(n^{3/2})\, n^{1/2} \log n + M(n) \log q)$, and the assertion about space is easy to check. □

As things now stand, distinct-degree factorization is the bottleneck in our general polynomial factoring algorithm. Furthermore, the bottleneck in our distinct-degree factorization algorithm is multi-point polynomial evaluation. We now indicate a direction in which further progress in reducing the time complexity of distinct-degree factorization—and hence the complexity of the general factoring problem—may be possible.

We define the *special multi-point polynomial evaluation problem* to be that of evaluating a “large” polynomial with “small” coefficients at a “small” number of “large” points. To be precise, let $f \in \mathbb{F}_q[x]$ have degree n, and let $R = \mathbb{F}_q[x]/(f)$. The special multi-point polynomial evaluation problem is that of computing $g(\alpha_1), \ldots, g(\alpha_m)$, where $m \le n^{1/2}$, $\alpha_1, \ldots, \alpha_m \in R$, and g is a polynomial of degree n over R whose coefficients each have canonical representatives of degree at most $n^{1/2}$.

Since the number of elements of $\mathbb{F}_q$ required to encode both the input and the output of this problem is only $O(n^{3/2})$, it is not a priori clear that this problem cannot be solved using, say, $O\tilde{}(n^{3/2})$ operations in $\mathbb{F}_q$. However, at present the best algorithms we know of take $O\tilde{}(n^2)$ operations in $\mathbb{F}_q$. All of the multi-point polynomial evaluation problems that arise in Algorithm 6.4 are of this special type. As the next theorem indicates, any improvement in solving this special problem will immediately lead to an improvement in the timing of distinct-degree factorization.

THEOREM 6.6. *Suppose that the special multi-point evaluation problem can be solved using $O\tilde{}(n^\kappa)$ operations in $\mathbb{F}_q$, with $3/2 \leq \kappa \leq 2$. Then Algorithm 6.4 can be implemented with $O\tilde{}(n^\kappa + n \log q)$ operations in $\mathbb{F}_q$.*

PROOF. Excluding steps 3, 4 and 6, and the invocations of Algorithm 6.2, Algorithm 6.4 uses $O\tilde{}(n^{3/2} + n \log q)$ operations in $\mathbb{F}_q$. Using the algorithm for special multi-point evaluation, steps 3, 4, and 6 can be implemented with $O\tilde{}(n^\kappa)$ operations in $\mathbb{F}_q$.

Now suppose we have a polynomial of degree n over R whose coefficients have canonical representatives of degree at most $n^{1/2}$, and we want to evaluate it at t points, where $t > n^{1/2}$. Using the algorithm for special multi-point evaluation, this can be done using $O\tilde{}((t/n^{1/2})\, n^\kappa)$ operations in $\mathbb{F}_q$ by processing the points $n^{1/2}$ at a time.

From the observation in the previous paragraph, if at iteration i of the loop in step 7, Algorithm 6.2 is invoked with a polynomial of degree n_i, then the number of operations in $\mathbb{F}_q$ used in this invocation is $O\tilde{}(n^{1/2} n_i^{\kappa-1/2})$. The sum of this quantity over all i is $O\tilde{}(n^\kappa)$, since $\kappa \geq 3/2$. Thus the total time of the algorithm is $O\tilde{}(n^\kappa + n \log q)$ operations in $\mathbb{F}_q$. $\square$

We conjecture that $\kappa < 2$ can be achieved.

7. Applications to other problems in finite fields

In this section, we use the techniques of the previous sections to obtain new algorithms for several problems in finite fields.

PROBLEM 7.1. *Given a polynomial of degree n over $\mathbb{F}_q$, determine if it is irreducible.*

PROBLEM 7.2. *Given a polynomial of degree n over $\mathbb{F}_q$, determine if all of its irreducible factors are distinct and have the same degree, and if so, determine the degree.*

Problem 7.2 is interesting since our algorithm for the equal-degree factorization problem is more efficient than our algorithm for factoring a general polynomial. Therefore, we want to be able to quickly test if we are in this special case, and if so, determine the degree of the irreducible factors since this is required as input to our equal-degree factoring algorithm. We use a well-known criterion for irreducibility (see Knuth 1981, Exercise 4.6.2–16).

FACT 7.3. *A polynomial $f \in \mathbb{F}_q[x]$ of degree n is irreducible if and only if $x^{q^n} - x \equiv 0 \bmod f$ and $\gcd(f, x^{q^{n/t}} - x) = 1$ for all primes t dividing n.*

The following is an easy generalization, the proof of which we leave to the reader.

FACT 7.4. *Let f be a polynomial of degree n over $\mathbb{F}_q$. The irreducible factors of f are all distinct and have the same degree if and only if $x^{q^n} - x \equiv 0 \bmod f$ and for all prime powers t dividing n, $\gcd(f, x^{q^{n/t}} - x)$ is either 1 or f. Moreover, if these conditions are met, then each irreducible factor of f has degree $d = n/\prod_t t$, where t ranges over the set of maximal prime power divisors of n such that $x^{q^{n/t}} - x \equiv 0 \bmod f$.*

We can use Algorithm 5.2 to compute $x^{q^m} \bmod f$ for the various values of m in the above two facts. In Fact 7.3, their number is bounded by the number of distinct prime divisors of n, which is $O(\log n/\operatorname{loglog} n)$ (see Hardy & Wright 1984, § 22.10). In Fact 7.4, this number is bounded by the number of prime power divisors of n, which is $O(\log n)$; however, we can achieve the same bound $O(\log n/\operatorname{loglog} n)$ by performing a binary search on the exponents of the prime factors. Recalling that C_T and C_S are time and space bounds for the modular composition problem (see Fact 5.1), we have the following result.

THEOREM 7.5. *Problems 7.1 and 7.2 can be solved by algorithms using*

$$O\Big((C_T(n) + M(n))(\log n)^2/\operatorname{loglog} n + M(n)\log q\Big)$$

operations in $\mathbb{F}_q$ and space for $O(C_S(n))$ elements of $\mathbb{F}_q$.

In particular, we have the following time and space bounds (time is measured in operations in $\mathbb{F}_q$ and space is measured in storage for elements of $\mathbb{F}_q$):

(i) time $O(n^2(\log n)^3 + n \log n \ \mathrm{loglog}\, n \ \log q)$ *and space* $O(n)$*;*

(ii) time $O(n^{1.7} + n \log n \ \mathrm{loglog}\, n \ \log q)$ *and space* $O(n \log n)$.

Related to the problem of testing irreducibility is that of constructing an irreducible polynomial of given degree n over $\mathbb{F}_q$. A probabilistic algorithm of Rabin (1980) for this problem uses $O\tilde{\ }(n^3 \log q)$ operations in $\mathbb{F}_q$, and another random method due to Ben-Or (1981) uses $O\tilde{\ }(n^2 \log q)$ operations, and Shoup (1993) presents an $O\tilde{\ }(n^2 + n \log q)$ method.

Problem 7.6. *Find a normal basis for* $\mathbb{F}_{q^n}$ *over* $\mathbb{F}_q$*, i.e., an* $\mathbb{F}_q$*-vector space basis of the form* $\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}$ *for* $\mathbb{F}_{q^n}$.

In von zur Gathen & Giesbrecht (1990), this problem is solved probabilistically with $O\tilde{\ }(n^2 \log q)$ operations in $\mathbb{F}_q$. The core of that algorithm is to compute $\beta_i = \alpha^{q^i} \in \mathbb{F}_{q^n}$ for $1 \le i < n$ and $\gcd(\sum_{0 \le i < n} \beta_i y^i, y^n - 1) \in \mathbb{F}_{q^n}[y]$, for $O(\log n)$ many random $\alpha \in \mathbb{F}_{q^n}$. Using Algorithm 3.1 to calculate these coefficients, we obtain the following improvement.

Theorem 7.7. *Problem 7.6 can be solved by a probabilistic algorithm using an expected number*

$$O(M(n^2)(\log n)^2 + M(n) \log q),$$

or

$$O((n^2(\log n)^2 + n \log q) \cdot \log n \ \mathrm{loglog}\, n)$$

of operations in $\mathbb{F}_q$ *and space for* $O(n^2)$ *elements of* $\mathbb{F}_q$.

Similarly, one can determine the *additive order* of an element of $\mathbb{F}_{q^n}$ with $O((n^2 \log n + n \log q) \cdot \log n \ \mathrm{loglog}\, n)$ operations in $\mathbb{F}_q$.

Interestingly, our algorithm is not the fastest way to compute what is usually called a trace, namely in a field extension.

Problem 7.8. *Compute the trace* $T_{\mathbb{F}_{q^n}/\mathbb{F}_q}(\alpha) = \sum_{0 \le i < n} \alpha^{q^i} \in \mathbb{F}_q$*, given* $f \in \mathbb{F}_q[x]$ *monic and irreducible of degree* n*, and* $\alpha \in \mathbb{F}_{q^n} = \mathbb{F}_q[x]/(f)$.

Write $T = T_{\mathbb{F}_{q^n}/\mathbb{F}_q}$ and $\xi = (x \bmod f) \in \mathbb{F}_{q^n}$, and let $\tilde{f} = \prod_{0 \le i < n}(1 - \xi^{q^i} y) \in \mathbb{F}_q[y]$ be the reverse of f. The expansion of the logarithmic derivative gives the following equation in $\mathbb{F}_{q^n}[[y]]$:

$$y\tilde{f}'/\tilde{f} = -\sum_{1 \le i} \sum_{0 \le j < n} \xi^{iq^j} y^i = -\sum_{1 \le i} T(\xi^i) y^i$$

(see Lidl & Niederreiter 1983, §5.2). The first n coefficients of the left-hand side are easy to calculate with a Newton iteration (see Borodin & Munro 1975, §4.4), and give $T(\xi^i)$ for $0 \leq i < n$. Since

$$T(\sum_{0\leq i<n} a_i\xi^i) = \sum_{0\leq i<n} a_i T(\xi^i)$$

for $a_0, \ldots, a_{n-1} \in \mathbb{F}_q$, T can then be evaluated as an inner product.

THEOREM 7.9. *Problem 7.8 can be solved with $O(M(n))$ operations in $\mathbb{F}_q$ and space for $O(n)$ elements of $\mathbb{F}_q$.*

8. Using classical arithmetic

We have presented a factoring algorithm that uses $O\tilde{}(n^2+n\log q)$ operations in $\mathbb{F}_q$. To achieve this running time, we have used asymptotically fast algorithms for polynomial arithmetic. One might ask of what value our methods are in the range of values for n in which classical polynomial arithmetic algorithms are faster than the asymptotically fast algorithms. We attempt to answer this question in this section. All running time estimates stated here assume classical arithmetic is used—that is, we assume that $M(n) = n^2$ and $MM(n) = n^3$.

The pioneering methods of Berlekamp (1970) and Cantor & Zassenhaus (1981) give factorization algorithms that use either $O(n^3 \log q)$ operations in $\mathbb{F}_q$ and space for $O(n)$ elements of $\mathbb{F}_q$, or $O(n^3 + n^2 \log q)$ operations in $\mathbb{F}_q$ and space for $O(n^2)$ elements of $\mathbb{F}_q$.

Using the above values for $M(n)$ and $MM(n)$ in Fact 5.1(iii), one finds that modular composition of polynomials over $\mathbb{F}_q$ of degree n can be done using classical arithmetic with $O(n^2(\log n)^3)$ operations in $\mathbb{F}_q$. In fact, this is a slight over-estimate, and the running time of this algorithm is actually $O((n\log n)^2)$ operations in $\mathbb{F}_q$. The space requirement is $O(n \log n)$ elements of $\mathbb{F}_q$.

As a consequence, by implementing the Frobenius map with modular composition, we can solve the distinct-degree factorization problem by successively computing the iterates of the Frobenius map one at a time using $O(n^3(\log n)^2 + n^2 \log q)$ operations in $\mathbb{F}_q$ and space for $O(n \log n)$ elements of $\mathbb{F}_q$. Algorithm 5.2 solves the problem of equal-degree factorization with $O(n^2(\log n)^3 + n^2 \log q)$ operations in $\mathbb{F}_q$ and space for $O(n \log n)$ elements of $\mathbb{F}_q$.

It is a bit difficult to interpret the meaning of these results. After all, an algorithm that uses only classical arithmetic might be so complicated in other respects that even for small inputs it is no faster than a different algorithm that uses asymptotically fast arithmetic. However, these results do suggest that our techniques could perhaps be used to implement an algorithm that is more efficient than other algorithms in its use of time or space for inputs of moderate size. For equal-degree factorization, preliminary experiments seem to indicate considerable improvements.

9. Deterministic equal-degree factorization

All of the algorithms we have discussed for squarefree and distinct-degree factorization are deterministic, and those for equal-degree factorization are probabilistic. In this section, we solve the latter problem with a *deterministic* procedure that combines the methods of the previous sections with those in Shoup (1991).

Suppose that the polynomial $f \in \mathbb{F}_q[x]$ to be factored is the product of r distinct monic irreducible polynomials, each of the same degree d, so that $n = \deg f = dr$. We denote by p the characteristic of $\mathbb{F}_q$, and write $q = p^k$. We assume that we have an element $\eta \in \mathbb{F}_q$ such that $\mathbb{F}_q = \mathbb{F}_p(\eta)$, i.e., the elements $1, \eta, \ldots, \eta^{k-1}$ are linearly independent over $\mathbb{F}_p$. In the usual representation of $\mathbb{F}_q$ as $\mathbb{F}_p[z]/(g)$, with $g \in \mathbb{F}_p[z]$ irreducible of degree k, we can take $\eta = (z \bmod g)$.

We describe and analyze a deterministic algorithm for this problem. The number of operations in $\mathbb{F}_q$ in one variant of this algorithm is

$$\tilde{O}(ndk + np^{1/2}\min\{dk, r\}), \tag{9.2}$$

and in a second variant, it is

$$\tilde{O}(n^2 + nk\min\{d, r\} + np^{1/2}\min\{dk, r\}). \tag{9.3}$$

Combining the second variant with one of our distinct-degree factorization algorithms, we obtain a general factorization algorithm that is deterministic and uses

$$\tilde{O}(n^2 + n^{3/2}k + n^{3/2}k^{1/2}p^{1/2})$$

operations in $\mathbb{F}_q$, since $\min\{d, r\} \le n^{1/2}$ and $\min\{dk, r\} \le (nk)^{1/2}$. Thus, in the special case where p is bounded, this algorithm uses $\tilde{O}(n^2 + n^{3/2}k)$ operations in $\mathbb{F}_q$.

At the end of this section, we briefly mention another variant that extracts a single irreducible factor of an arbitrary polynomial using

$$\tilde{O}(n^2 + nk + np^{1/2})$$

operations in $\mathbb{F}_q$. Thus, when p is bounded, this is $\tilde{O}(n^2 + nk)$, which is as fast as our probabilistic algorithm to within logarithmic factors.

As usual, we write $f = f_1 \cdots f_r$ and $R = \mathbb{F}_q[x]/(f)$. Under the Chinese Remainder isomorphism of R onto $\bigoplus_{1 \le i \le r} \mathbb{F}_{q^d}$, we identify an element $\alpha \in R$ with its component representation $(\alpha_1, \ldots, \alpha_r)$, where each α_i is in $\mathbb{F}_{q^d}$, and we call α_i the ith component of α.

The *(absolute) Berlekamp subalgebra* A of R consists of those elements in R whose components lie in the prime field $\mathbb{F}_p$. A set $S \subseteq R$ is called a *separating set* if for all i, j with $1 \le i < j \le r$ there exists $\alpha \in S$ whose ith and jth components are not equal.

Computing a separating set in A lies at the heart of all known deterministic factoring algorithms. Consider two distinct factors f_i and f_j. If $S \subseteq A$ is a separating set, then we know that for some $\alpha \in S$ with component representation $(\alpha_1, \ldots, \alpha_r)$, we have $\alpha_i \neq \alpha_j$. Therefore, the polynomial $\check{\alpha} - \alpha_i \in \mathbb{F}_q[x]$ is divisible by f_i but not f_j, and its gcd with f splits f_i and f_j apart. Now, given α, we do not know the values of its components, but we do know that they lie in $\mathbb{F}_p$, so by trying all values of a in $\mathbb{F}_p$, we can find a polynomial $\check{\alpha} - a$ that is divisible by one of f_i or f_j but not both.

The procedure just described requires time proportional to p in the worst case. If p is large we can speed this process up as follows. Compute $\beta = (\alpha - z)^{(p-1)/2}$ for $z = 0, 1, 2, \ldots$. The ith component of β is $\chi(\alpha_i - z)$, where χ is the quadratic character on $\mathbb{F}_p$. Thus, if $\chi(\alpha_i - z) \neq \chi(\alpha_j - z)$, then either $\check{\beta}$ or $\check{\beta} - 1$ is divisible by f_i or f_j but not both. It was shown in Shoup (1990) that for any pair of distinct elements $a, b \in \mathbb{F}_p$, the least nonnegative integer z such that $\chi(a - z) \neq \chi(b - z)$ is bounded by $O(p^{1/2} \log p)$; this bound was subsequently improved in Shparlinski (1992) to $O(p^{1/2})$. Thus, the number of values of z we need to try in order to separate f_i from f_j is $O(p^{1/2})$.

If p is small and the size of S is not too large, we can use these observations to obtain an efficient algorithm to completely factor f. Howevever, when p is small it is the computation of a separating set in A that is actually the bottleneck in deterministic factoring algorithms. Our main technical contribution in this section is a faster method for computing a separating set.

Let $T \colon R \to R$ be the trace map that sends $\alpha \in R$ to $\alpha + \alpha^p + \cdots + \alpha^{p^{k-1}}$.

THEOREM 9.1. *Let*

$$H = (y - \xi)(y - \xi^q) \cdots (y - \xi^{q^{d-1}}) = h_0 + h_1 y + \cdots + h_{d-1} y^{d-1} + y^d \in R[y].$$

Then the set

$$S = \{T(\eta^v h_u) : 0 \le v < k,\ 0 \le u < d\} \subseteq A$$

is a separating set.

PROOF. Let B be the subring of R consisting of those elements whose components lie in $\mathbb{F}_q$. We claim the following two properties:

1. Each h_u lies in B.

2. $\{h_0, \ldots, h_{d-1}\}$ is a separating set.

Let $(h_{u,1}, \ldots, h_{u,r})$ be the component representation of h_u for $0 \le u < d$, and let $f_i = x^d + \sum_{0 \le u < d} a_{u,i} x^u$ with each $a_{u,i}$ in $\mathbb{F}_q$. Let $(\xi_1, \ldots, \xi_r)$ be the component representation of ξ. Then we have $f_i = \prod_{0 \le u < d}(x - \xi_i^{q^u})$, from which it follows that $h_{u,i} = a_{u,i}$ for $0 \le u < d$ and $1 \le i \le r$. The claims now easily follow.

The map T acts component-wise on B as the trace $T_{\mathbb{F}_q/\mathbb{F}_p}$ from $\mathbb{F}_q$ to $\mathbb{F}_p$. So it is clear that $S \subseteq A$. Given i and j with $1 \le i < j \le r$, we choose h_u such that $h_{u,i} \ne h_{u,j}$. Then, since $T_{\mathbb{F}_q/\mathbb{F}_p}$ is an $\mathbb{F}_p$-linear map from $\mathbb{F}_q$ onto $\mathbb{F}_p$, and since $1, \eta, \ldots, \eta^{k-1}$ form a basis for $\mathbb{F}_q$ over $\mathbb{F}_p$, we see that for some $v < k$ we have $T_{\mathbb{F}_q/\mathbb{F}_p}(\eta^v h_{u,i}) \ne T_{\mathbb{F}_q/\mathbb{F}_p}(\eta^v h_{u,j})$. For this u and v, the ith and jth components of $T(\eta^v h_u)$ are not equal, and hence S is a separating set. □

The observation that the coefficients of H form a separating set was made in Shoup (1990). We now address the complexity of computing the elements in this separating set.

THEOREM 9.2. *Let $H \in R[y]$ be the polynomial in Theorem 9.1.*

(i) There is a deterministic algorithm that computes the coefficients of H using

$$O(M(dn)\log d + M(n) d \log q)$$

operations in $\mathbb{F}_q$, and space for $O(dn)$ elements of $\mathbb{F}_q$.

(ii) There is a deterministic algorithm that computes the coefficients of H using

$$O(M(dn) r \log d + M(n)\log q)$$

operations in $\mathbb{F}_q$, and space for $O(dn)$ elements of $\mathbb{F}_q$.

(iii) There is a deterministic algorithm that takes as input an element $\alpha \in R$ and computes $T(\eta^v \alpha)$ for $0 \le v < k$ using

$$O(M(n)\log q + nM(k)\log k)$$

operations in $\mathbb{F}_q$, and space for $O(kn)$ elements of $\mathbb{F}_q$.

PROOF. We can compute $\xi, \xi^q, \ldots, \xi^{q^{d-1}}$, where $\xi = (x \bmod f) \in R$, by repeated squaring using $O(d \log q)$ operations in R, and hence $O(M(n) d \log q)$ operations in $\mathbb{F}_q$. Alternatively, we can just compute ξ^q using repeated squaring, and then apply Algorithm 3.1. We can form the product $\prod_{0 \le u < d}(y - \xi^{q^u})$ $\in R[y]$ using a divide-and-conquer algorithm with $O(M(dn) \log d)$ operations in $\mathbb{F}_q$. This proves (i) and (ii). For (iii), we compute the quantities

$$\eta_w = \eta^{p^w} \text{ and } \alpha_w = \alpha^{p^w} \quad \text{for } 0 \le w < k$$

by repeated squaring using $O(M(n) \log q)$ operations in $\mathbb{F}_q$. Then for $0 \le v < k$, we have

$$T(\eta^v \alpha) = \sum_{0 \le w < k} \eta_w^v \alpha_w.$$

The next section provides an algorithm that given the η_w's and the α_w's computes all of these *weighted power sums* with $O(M(k) \log k)$ operations in R, and hence $O(nM(k) \log k)$ operations in $\mathbb{F}_q$, since the multiplications involve at most one element outside of $\mathbb{F}_q$ (Theorem 10.4). That algorithm also uses space for $O(k)$ elements of R. $\square$

Part (iii) of this theorem says that for a given coefficient h_u of H, we can compute $T(\eta^v h_u)$ for $0 \le v < k$ using $O\tilde{~}(n \log q)$ operations in $\mathbb{F}_q$. To compute the entire separating set S in Theorem 9.1, we could repeat this for each of the d coefficients of H. This would require $O\tilde{~}(nd \log q)$ operations in $\mathbb{F}_q$, which for fixed p is optimal up to logarithmic factors, but is more than we can allow if we want to achieve the running time bound (9.3). However, as we shall see below, it is possible to avoid computing S in its entirety, and it is sufficient to consider $\min\{d, r\}$ coefficients of H. Thus, we can compute the required portion of S using only $O\tilde{~}(\min\{d, r\} n \log q)$ operations in $\mathbb{F}_q$ (plus $O\tilde{~}(dn)$ to determine that portion).

Before we describe our deterministic equal-degree factorization algorithm, we need a few more observations. A set $U \subseteq \mathbb{F}_q[x]$ of monic nonconstant polynomials is called a *refinement* of f if $f = \prod_{g \in U} g$. We use the following two operations.

1. *Refine*(U, α), where U is a refinement of f and $\alpha \in R$. This replaces U by the refinement U' obtained in the following way: for each $g \in U$, if $g_1 = \gcd(g, \check{\alpha})$ is a trivial divisor of g, put g in U'; otherwise, put g_1 and g/g_1 in U'.

2. *Useful*(U, α), where U is a refinement of f and $\alpha \in R$. This evaluates to *true* if $(\check{\alpha} \text{ rem } g) \notin \mathbb{F}_q$ for some $g \in U$; otherwise, it evaluates to *false*.

The following lemma addresses the cost of these operations.

LEMMA 9.3. *Operation Refine*(U, α) *can be performed using* $O(M(n) \log n)$ *operations in* $\mathbb{F}_q$ *and space for* $O(n)$ *elements of* $\mathbb{F}_q$. *The predicate Useful*(U, α) *can be evaluated using* $O(M(n) \log r)$ *operations in* $\mathbb{F}_q$ *and space for* $O(n)$ *elements of* $\mathbb{F}_q$.

PROOF. To implement either of these operations, we begin by computing $\check{\alpha}$ rem g for all $g \in U$. Since $\#U \leq r$, this can be done with $O(M(n) \log r)$ operations in $\mathbb{F}_q$ and space for $O(n)$ elements in $\mathbb{F}_q$ in a manner similar to the proof of Lemma 2.1. This gives us the bound for computing *Useful*(U, α). The algorithm for *Refine*(U, α) then proceeds by computing $\gcd(\check{\alpha} \text{ rem } g, g)$ for all $g \in U$. This takes $O(M(n) \log n)$ operations in $\mathbb{F}_q$ and space for $O(n)$ elements in $\mathbb{F}_q$. $\square$

ALGORITHM 9.4. *Deterministic equal-degree factorization.*
Input: *A squarefree monic polynomial* f *over* $\mathbb{F}_q$ *of degree* n, *a positive integer* d, *and an element* $\eta \in \mathbb{F}_q$. *It is assumed that* f *is the product of* r *irreducible factors, each of degree* d *(so that* $n = rd$*), and that* $\mathbb{F}_q = \mathbb{F}_{p^k} = \mathbb{F}_p(\eta)$.
Output: *The set of monic irreducible factors of* f.

1. *In the notation (3.1), compute the coefficients of*
$$H = (y - \xi)(y - \xi^q) \cdots (y - \xi^{q^{d-1}})$$
$$= h_0 + h_1 y + \cdots + h_{d-1} y^{d-1} + y^d \in R[y].$$
2. *Set* $U = \{f\}$.
3. *For* $u = 0, \ldots, d-1$ *do step 4.*
4. *If Useful*(U, h_u), *then do steps 5 and 6.*
5. *Compute* $(\alpha_0, \alpha_1, \ldots, \alpha_{k-1}) = (T(h_u), T(\eta h_u), \ldots, T(\eta^{k-1} h_u))$.
6. *For* $v = 0, \ldots, k-1$ *do steps 7 and 8.*
7. *Set* $z = 0$.
8. *While Useful*(U, α_v) *do steps 9 through 11.*
9. *Refine*$(U, \alpha_v - z)$.
10. *If* $p \neq 2$, *then Refine*$(U, (\alpha_v - z)^{(p-1)/2} - 1)$.
11. *Replace* z *by* $z + 1$.
12. *Return* U.

THEOREM 9.5. *Algorithm 9.4 works correctly as specified. If step 1 is implemented using Theorem 9.2(i), then it uses*

$$O\Big(M(dn)\log d + M(n)d\log(rq) \\ + \min\{d,r\}[nM(k)\log k + M(n)k\log r] + \min\{dk,r\}p^{1/2}M(n)\log(np)\Big)$$

or, more briefly,

$$O\tilde{}(ndk + np^{1/2}\min\{dk,r\})$$

operations in $\mathbb{F}_q$. *If step 1 is implemented using Theorem 9.2(ii), then it uses*

$$O\Big(\min\{d,r\}[nM(k)\log k + M(n)k\log(pr)] \\ + M(dn)r\log d + \min\{dk,r\}p^{1/2}M(n)\log(np)\Big)$$

or, more briefly,

$$O\tilde{}(n^2 + nk\min\{d,r\} + np^{1/2}\min\{dk,r\})$$

operations in $\mathbb{F}_q$. *In either case, it uses space for* $O((d+k)n)$ *elements of* $\mathbb{F}_q$.

PROOF. It follows immediately from Theorem 9.1 that the algorithm would work correctly if the two *Useful* tests were left out. However, their insertion only avoids unnecessary work; this proves correctness.

To estimate the running time, notice that if *Useful*(U, h_u) at step 4 is true, then execution of steps 5 and 6 produces a proper refinement of U; hence, steps 5 and 6 are executed at most $\min\{d,r\}$ times. Similarly, the inner loop at steps 9 through 11 is executed at most $O(p^{1/2}\min\{dk,r\})$ times.

The statements about the running time follow from these observations, along with the estimates from Theorem 9.2. The space bound is clear. □

REMARK. Notice that Algorithm 9.4, using Theorem 9.2(ii) to implement step 1, discovers a nontrivial factor after executing just $O\tilde{}(n^2 + nk + np^{1/2})$ operations in $\mathbb{F}_q$. This implies that there is a deterministic algorithm that takes as input an arbitrary polynomial $f \in \mathbb{F}_q[x]$ of degree n, and produces as output a single irreducible factor of f using $O\tilde{}(n^2 + nk + np^{1/2})$ operations in $\mathbb{F}_q$.

10. Generalized power sums

The algorithm in the proof of Theorem 9.2(iii) uses as a subroutine an algorithm for the following general problem: given $\alpha_0, \ldots, \alpha_{k-1}$ and $\beta_0, \ldots, \beta_{k-1}$ in a ring R, compute the *weighted power sums*

$$\gamma_\ell = \sum_{0 \le i < k} \alpha_i^\ell \beta_i \qquad \text{for } 0 \le \ell < k.$$

In this section, R is an arbitrary commutative ring with unity. This problem is equivalent to computing the matrix-vector product $V^T b$, where $V \in R^{k \times k}$ is the Vandermonde matrix with $V_{ij} = \alpha_{i-1}^{j-1}$ and $b = (\beta_0, \ldots, \beta_{k-1})^T \in R^{k \times 1}$. We shall present an algorithm for this problem that uses $O(M(k) \log k)$ operations in R, and space for $O(k)$ elements in R.

An algorithm for this problem is described in Canny *et al.* (1989). It uses $O(M(k) \log k)$ operations in R, and it is possible—using Lemma 2.1—to implement this algorithm so that it uses space for $O(k)$ elements in R. This algorithm works by first computing $c = V^{-1} b$, and then applying the Hankel matrix $V^T V$ to c. In their application, R is a field and the α_i's are distinct. The use of this algorithm over an arbitrary ring R is hindered by the fact that it performs several divisions by elements in R, and it is not clear that these can easily be avoided.

There are general methods by which one can transform an algebraic circuit for computing the matrix-vector product Mb into an algebraic circuit of about the same size for computing $M^T b$ (Baur & Strassen 1983, Kaminski *et al.* 1988). Since computing Vb is known to take $O(M(k) \log k)$ operations in R, the same bound applies to computing $V^T b$. The algorithms that result from these general transformations require space proportional to their running times.

Either of these methods would have sufficed to obtain the running time bound in Theorem 9.2(iii); however, our algorithm below is still perhaps of interest in itself, since it avoids divisions, requires space for only $O(k)$ elements of R, and has a fairly natural description. We start with an algorithm for computing the first nontrivial elements of a linearly recurrent sequence.

LEMMA 10.1. *Suppose we are given elements $c_1, \ldots, c_k$ and $\gamma_0, \ldots, \gamma_{k-1}$ in R. For $\ell \ge k$ let γ_ℓ be defined by the recurrence*

$$\gamma_\ell + c_1 \gamma_{\ell-1} + \cdots + c_k \gamma_{\ell-k} = 0. \qquad (10.4)$$

Then we can compute $\gamma_k, \ldots, \gamma_{2k-1}$ using $O(M(k))$ operations in R and space for $O(k)$ elements in R.

PROOF. Define the polynomials

$$G = \sum_{0\le j\le k} c_j x^j, \qquad U = \sum_{0\le j<k} \gamma_j x^j, \qquad V = \sum_{0\le j<k} \gamma_{k+j} x^j,$$

where c_0 is taken to be 1. Now consider the products

$$GU = \sum_{0\le j<2k} u_j x^j, \qquad GV = \sum_{0\le j<2k} v_j x^j.$$

Clearly (10.4), for $k \le \ell < 2k$, is equivalent to the condition that

$$G\cdot(U + x^k V) \text{ rem } x^{2k}$$

have degree less than k. Let $GU = F_0 + x^k F_1$, where F_0 and F_1 are polynomials of degree at most $k-1$. Then we can rewrite this as

$$GV \equiv -F_1 \bmod x^k.$$

Let $H \in R[x]$ have degree less than k and satisfy $HG \equiv 1 \bmod x^k$; this exists since $c_0 = 1$. Then we have

$$V \equiv -HF_1 \bmod x^k.$$

Thus we can compute H by a Newton iteration, find F_1 from GU, and obtain the desired outputs from $V = -HF_1 \text{ rem } x^k$. All this can be done within the stated time and space bounds. □

Our algorithm for computing weighted power sums is based on the following lemma.

LEMMA 10.2. *Let $\alpha_0, \ldots, \alpha_{k-1}$ and $\beta_0, \ldots, \beta_{k-1}$ be in R. Define $\gamma_\ell \in R$ for $\ell \ge 0$, $G \in R[x]$, and $c_0, \ldots, c_k \in R$ by*

$$\gamma_\ell = \sum_{0\le i<k} \alpha_i^\ell \beta_i, \quad G = \prod_{0\le j<k} (1-\alpha_i x) = \sum_{0\le j\le k} c_j x^j \in R[x].$$

Then

$$\gamma_\ell + c_1\gamma_{\ell-1} + \cdots + c_k\gamma_{\ell-k} = 0 \qquad \textit{for all } \ell \ge k.$$

PROOF. Let

$$\tilde{G} = \sum_{0\le j\le k} c_j x^{k-j} = \prod_{0\le j<k} (x-\alpha_i).$$

Then for $\ell \geq k$, we have

$$\begin{aligned}\sum_{0 \leq j \leq k} c_j \gamma_{\ell-j} &= \sum_{0 \leq j \leq k} c_j \sum_{0 \leq i < k} \alpha_i^{\ell-j} \beta_i \\ &= \sum_{0 \leq i < k} \alpha_i^{\ell-k} \beta_i \sum_{0 \leq j \leq k} c_j \alpha_i^{k-j} = \sum_{0 \leq i < k} \alpha_i^{\ell-k} \beta_i \tilde{G}(\alpha_i) = 0. \qquad \square\end{aligned}$$

ALGORITHM 10.3. *Weighted power sums.*
Input: *Elements* $\alpha_0, \ldots, \alpha_{k-1}$ *and* $\beta_0, \ldots, \beta_{k-1}$ *in* R*, where* k *is a power of 2.*
Output: $\gamma_0, \ldots, \gamma_{k-1}$ *and* $c_0, \ldots, c_k$*, as defined in Lemma 10.2.*

1. *If* $k = 1$*, then return* β_0 *and* $(1 - \alpha_0 x)$*.*

2. *Otherwise, divide the problem into two pieces of size* $k/2$*, and recursively compute the following quantities:*

   (i) $\gamma'_\ell = \sum_{0 \leq i < k/2} \alpha_i^\ell \beta_i$ *for* $0 \leq \ell < k/2$,

   (ii) *the coefficients of* $G' = (1 - \alpha_0 x) \cdots (1 - \alpha_{k/2-1} x)$

   (iii) $\gamma''_\ell = \sum_{k/2 \leq i < k} \alpha_i^\ell \beta_i$ *for* $0 \leq \ell < k/2$,

   (iv) *the coefficients of* $G'' = (1 - \alpha_{k/2} x) \cdots (1 - \alpha_{k-1} x)$

3. *Extend the sequences* $(\gamma'_0, \ldots, \gamma'_{k/2-1})$ *and* $(\gamma''_0, \ldots, \gamma''_{k/2-1})$ *to the corresponding sequences of length* k *using the coefficients of* G' *and* G''*, and the algorithm of Lemma 10.1.*

4. *For* $0 \leq \ell < k$*, set* $\gamma_\ell = \gamma'_\ell + \gamma''_\ell$*, and compute the coefficients of* $G = G'G''$*.*

5. *Return* $\gamma_0, \ldots, \gamma_{k-1}$ *and the coefficients of* G*.*

THEOREM 10.4. *Algorithm 10.3 works correctly as specified, and uses* $O(M(k) \cdot \log k)$ *operations in* R *and space for* $O(k)$ *elements in* R*.*

The proof is straightforward. In this algorithm, we assumed for simplicity that k is a power of 2. Of course this restriction can easily be removed, either by modifying the algorithm slightly, or by padding the input.

Acknowledgements

We are indebted to Erich Kaltofen for sharing his idea of the polynomial representation of the Frobenius map with us, and for the many enthusiastic discussions we had during his sabbatical visit at Toronto.

This work was supported by ITRC and the Natural Sciences and Engineering Research Council of Canada. An Extended Abstract appeared in Proc. 24th Ann. ACM Symp. Theory of Computing, Victoria BC, 1992, 97–105.

References

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

A. Arwin. Über Kongruenzen von dem fünften und höheren Graden nach einem Primzahlmodulus. *Arkiv för matematik, astronomi o. fysik* **14** (1918), 1–46.

L. Babai, E. M. Luks, and Á. Seress. Fast management of permutation groups. In *29th Annual Symposium on Foundations of Computer Science*, 272–282, 1988.

W. Baur and V. Strassen. The complexity of computing partial derivatives. *Theoret. Comput. Sci.* **22** (1983), 317–330.

M. Ben-Or. Probabilistic algorithms in finite fields. In *22nd Annual Symposium on Foundations of Computer Science*, 394–398, 1981.

E. R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, 1968.

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

A. Borodin and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems*. American Elsevier, 1975.

R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *J. Assoc. Comput. Mach.* **25** (1978), 581–595.

J. Buchmann. Complexity of algorithms in algebraic number theory. In *Number Theory. Proc. First Conf. Canadian Number Theory Assoc.*, 37–53. Walter de Gruyter, 1990.

M. C. R. Butler. On the reducibility of polynomials over a finite field. *Quart. J. Math., Oxford Ser. (2)* **5** (1954), 102–107.

P. Camion. Improving an algorithm for factoring polynomials over a finite field and constructing large irreducible polynomials. *IEEE Trans. Inform. Theory* **IT-29** (1983), 378–385.

J. F. Canny, E. Kaltofen, and L. Yagati. Solving systems of non-linear polynomial equations faster. In *Proc. Int. Symp. on Symbolic and Algebraic Comp.*, 121–128, 1989.

D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta. Inf.* **28** (1991), 693–701.

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

D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comp.* **9** (1990), 23–52.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press, 1989.

J. von zur Gathen. Irreducibility of multivariate polynomials. *J. Computer System Sciences* **31** (1985), 225–264.

J. von zur Gathen. Factoring polynomials and primitive elements for special primes. *Theoret. Comput. Sci.* **52** (1987), 77–89.

J. von zur Gathen and M. Giesbrecht. Constructing normal bases in finite fields. *J. Symb. Comp.* **10** (1990), 547–570.

J. von zur Gathen and G. Seroussi. Boolean circuits versus arithmetic circuits. *Inform. and Comput.* **91** (1991), 142–154.

G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, fifth edition, 1984.

E. Kaltofen. Polynomial factorization 1982–1986. In Computers in Mathematics*, ed. D. V. Chudnovsky, R. D. Jenks, Lecture Notes in Pure and Applied Mathematics, vol. 125*, 285–309, 1990.

M. Kaminski, D. G. Kirkpatrick, and N. H. Bshouty. Addition requirements for matrix and transposed matrix products. *J. of Algorithms* **9** (1988), 354–364.

D. E. Knuth. *The Art of Computer Programming*, vol. 2. Addison-Wesley, second edition, 1981.

R. Lidl and H. Niederreiter. *Finite Fields.* Addison-Wesley, 1983.

R. J. McEliece. Factorization of polynomials over finite fields. *Math. Comp.* **23** (1969), 861–867.

A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. Subgroup refinement algorithms for root finding in $GF(q)$. *SIAM J. Comput.* **21** (1992), 228–239.

M. Mignotte and C. Schnorr. Calcul des racines d-ièmes dans un corps fini. *C. R. Acad. Sci. Paris* **290** (1988), 205–206.

R. T. Moenck. On the efficiency of algorithms for polynomial factoring. *Math. Comp.* **31** (1977), 235–250.

A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Advances in Cryptology, Proceedings of Eurocrypt 84*, 224–314. Springer-Verlag, 1985.

M. O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput.* **9** (1980), 273–280.

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

A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing* **7** (1971), 281–292.

V. Shoup. On the deterministic complexity of factoring polynomials over finite fields. *Inform. Process. Lett.* **33** (1990), 261–267.

V. Shoup. A fast deterministic algorithm for factoring polynomials over finite fields of small characteristic. In *Proc. Int. Symp. on Symbolic and Algebraic Comp.*, 14–21, 1991.

V. Shoup. Fast construction of irreducible polynomials over finite fields. In *Proc. IEEE Symp. on Discrete Algorithms*, Austin, TX, 1993.

V. Shoup and R. Smolensky. An algorithm for modular composition. Preprint, 1992.

I. E. Shparlinski. *Computational problems in finite fields.* Kluwer, 1992. To appear.

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

A. Thiong ly. A deterministic algorithm for factorizing polynomials over extensions $GF(p^m)$ of $GF(p)$, p a small prime. *J. of Information and Optimization Sciences* **10** (1989), 337–344.

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

Manuscript received 20 November 1991

JOACHIM VON ZUR GATHEN
VICTOR SHOUP
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4, Canada
gathen@theory.toronto.edu
shoup@theory.toronto.edu