Discrete logarithms in finite fields and their cryptographic significance

A. M. Odlyzko

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Given a primitive element g of a finite field $GF(q)$, the discrete logarithm of a nonzero element $u \in GF(q)$ is that integer k, $1 \leq k \leq q-1$, for which $u = g^k$. The well-known problem of computing discrete logarithms in finite fields has acquired additional importance in recent years due to its applicability in cryptography. Several cryptographic systems would become insecure if an efficient discrete logarithm algorithm were discovered. This paper surveys and analyzes known algorithms in this area, with special attention devoted to algorithms for the fields $GF(2^n)$. It appears that in order to be safe from attacks using these algorithms, the value of n for which $GF(2^n)$ is used in a cryptosystem has to be very large and carefully chosen. Due in large part to recent discoveries, discrete logarithms in fields $GF(2^n)$ are much easier to compute than in fields $GF(p)$ with p prime. Hence the fields $GF(2^n)$ ought to be avoided in all cryptographic applications. On the other hand, the fields $GF(p)$ with p prime appear to offer relatively high levels of security.

Discrete logarithms in finite fields and their cryptographic significance

A. M. Odlyzko

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

The multiplicative subgroup of any finite field $GF(q)$, q a prime power, is cyclic, and the elements $g \in GF(q)$ that generate this subgroup are referred to as *primitive elements*. Given a primitive element $g \in GF(q)$ and any $u \in GF(q)^* = GF(q) - \{0\}$, the *discrete logarithm* of u with respect to g is that integer k, $0 \le k \le q-1$, for which

$$u = g^k \ .$$

We will write $k = \log_g u$. The discrete logarithm of u is sometimes referred to as the index of u.

Aside from the intrinsic interest that the problem of computing discrete logarithms has, it is of considerable importance in cryptography. An efficient algorithm for discrete logarithms would make several authentication and key-exchange systems insecure. This paper briefly surveys (in Section 2) these cryptosystems, and then analyzes the known algorithms for computing discrete logarithms. As it turns out, some of them, including the most powerful general purpose algorithm in this area, have not been analyzed in complete detail before. Moreover, some of the analyses in the literature deal only with fields $GF(p)$, where p is a prime. In cryptographic applications, on the other hand, attention has been focused on the fields $GF(2^n)$, since arithmetic in them is much easier to implement, with respect to both software and hardware. Therefore we concentrate on the fields $GF(2^n)$.

Several proposed algorithms for computing discrete logarithms are known. We briefly discuss most of them (including some unsuccessful ones) in Section 3. In Section 4 we present the most powerful general purpose algorithm that is known today, called the index-calculus algorithm, and analyze its asymptotic performance. Recently a dramatic improvement in its performance in fields $GF(2^n)$ was made by Coppersmith [18,19], and we discuss it in detail. In Section 5 we discuss several technical issues that are important to the performance of the index-calculus algorithm, such as rapid methods to solve the systems of linear equations that arise in it. In that section we also present several suggested modifications to the Coppersmith algorithm which appear to be unimportant asymptotically, but are of substantial importance in practice. We discuss them in order to obtain a reasonable estimate of how fast this algorithm could be made to run in practice. In Section 6 we estimate the running time of that algorithm for fields $GF(2^n)$ that might actually be used in cryptography. In Section 7 we briefly discuss the performance of the index-calculus algorithms in fields $GF(p)$ for p a prime. Finally, we discuss the implications of these algorithms for cryptography in Section 8. It turns out, for example, that the MITRE scheme [38,59] and the Hewlett-Packard chip [69], both of which use the field $GF(2^{127})$, are very insecure. Depending on the level of security that is desired, it seems that fields $GF(2^n)$ to be used ought to have n large, no smaller than 800 and preferably at least 1500. Furthermore, these values of n have to be very carefully chosen. On the other hand, it appears at this moment that the fields $GF(p)$, where p is a prime, offer a much higher level of security, with $p \gtrsim 2^{500}$ adequate for many applications and $p \gtrsim 2^{1000}$ being sufficient even for extreme situations. The fields $GF(p)$ appear at this moment to offer security comparable to that of the RSA scheme with modulus of size p.

It has to be stressed that this survey presents the current state of the art of computing discrete logarithms. Since the state of the art has been advancing very rapidly recently, this paper has already gone through several revisions. The most important of the new developments has certainly been the Coppersmith breakthrough in fields $GF(2^n)$. Even more recently, there has been much less spectacular but still important progress in fields $GF(p)$, which is briefly described in Section 7, and in methods for dealing with sparse systems of equations, which are discussed in Section 5, and which are crucial for the index-calculus algorithms. It is quite likely that further progress will take place in discrete logarithm algorithms and so the cryptographic schemes described below will require the use of even larger fields than are being recommended right now.

2. Cryptographic systems related to discrete logarithms

One of the first published cryptosystems whose security depends on discrete logarithms being difficult to compute appears to be an authentication scheme. In many computer systems, users' passwords are stored in a special file, which has the disadvantage that anyone who gets access to that file is able to freely impersonate any legitimate user. Therefore that file has to be specially protected by the operating system. It has been known for a long time (cf. [54]) that one can eliminate the need for any secrecy by eliminating the storage of passwords themselves. Instead, one utilizes a function *f* that is hard to invert (i.e., such that given a *y* in the range of *f*, it is hard to find an *x* in the domain of *f* such that $f(x) = y$) and creates a file containing pairs $(i, f(p_i))$, where *i* denotes a user's login name and p_i the password of that user. This file can then be made public. The security of this scheme clearly depends on the function *f* being hard to invert. One early candidate for such a function was discrete exponentiation; a field $GF(q)$ and a primitive element $g \in GF(q)$ are chosen (and made public), and for *x* an integer, one defines

$$f(x) = g^x .$$

Anyone trying to get access to a computer while pretending to be user *i* would have to find p_i knowing only the value of g^{p_i}; i.e., he would have to solve the discrete logarithm problem in the field $GF(q)$.

Public key cryptography suffers from the defect that the systems that seem safe are rather slow. This disadvantage can be overcome to a large extent by using a public key cryptosystem only to distribute keys for a classical cryptosystem, which can then be used to transmit data at high speeds. Diffie and Hellman [23] have invented a key-exchange system based on exponentiation in finite fields. (This apparently was the very first public key cryptosystem that was proposed.) In it, a finite field $GF(q)$ and a primitive element $g \in GF(q)$ are chosen and made public. Users *A* and *B*, who wish to communicate using some standard encryption method, such as DES, but who do not have a common key for that system, choose random integers *a* and *b*, respectively, with $2 \le a, b \le q-2$. Then user *A* transmits g^a to *B* over a public channel, while user *B* transmits g^b to *A*. The common key is then taken to be g^{ab}, which *A* can compute by raising the received g^b to the *a* power (which only he knows), and which *B* forms by raising g^a to the *b* power. It is clear that an efficient discrete logarithm algorithm would make this scheme insecure, since the publicly transmitted g^a would enable the cryptanalyst to determine *a*, and he could then determine the key used by *A* and *B*. Diffie and Hellman [23] have even conjectured that breaking their scheme is equivalent in difficulty to computing discrete logarithms. This conjecture remains unproved, and so we cannot exclude the possibility that there might be some way to generate g^{ab} from knowledge of g^a and g^b only, without computing either *a* or *b*, although it seems unlikely that such a method exists.

The Diffie-Hellman key-exchange scheme seems very attractive, and it has actually been implemented in several systems, such as a MITRE Corp. system [38,59]. Moreover, Hewlett-Packard has built a special purpose VLSI chip which implements this scheme [69]. However, these implementations have turned out to be easily breakable. It appears possible, though, to build a Diffie - Hellman scheme that is about as secure as an RSA scheme of comparable key size. This will be discussed at some length in Section 8.

Systems that use exponentiation in finite fields to transmit information have also been proposed. One is based on an idea due to Shamir [37; pp. 345-346] and has been advocated in the context of discrete exponentiation by Massey and Omura [63]. For example, suppose user *A* wishes to send a message *m* (which we may regard as a nonzero element of the publicly known field $GF(q)$) to user *B*. Then *A* chooses a random integer *c*, $1 \le c \le q-1$, $(c, q-1) = 1$, and transmits $x = m^c$ to *B*. User *B* then chooses a random integer *d*, $1 \le d \le q-1$, $(d, q-1) = 1$, and transmits $y = x^d = m^{cd}$ to *A*. User *A* now forms $z = y^{c'}$ where $cc' \equiv 1 \pmod{q-1}$, and transmits *z* to *B*. Since

$$z = y^{c'} = m^{cdc'} = m^d ,$$

B only has to compute $z^{d'}$ to recover *m*, where $dd' \equiv 1 \pmod{q-1}$, since

$$z^{d'} = m^{dd'} = m .$$

In this scheme it is again clear that an efficient method for computing discrete logarithms over $GF(q)$ would enable a cryptanalyst to recover the plaintext message *m* from the transmitted ciphertext messages m^c, m^{cd}, and m^d.

Another scheme for transmission of information has been proposed by T. ElGamal [26] and is in essence a variant of the Diffie-Hellman key distribution scheme. User A publishes a public key $g^a \in GF(q)$, where the field $GF(q)$ and a primitive root *g* are known (either they are also published by A or else they are used by everyone in a given system), but keeps *a* secret. User B, who wishes to send $m \in GF(q)$ to A, selects *k* at random, $1 \le k \le q-2$ (a different *k* has to be chosen for each *m*) and transmits the pair (g^k, mg^{ak}) to A. User A knows *a* and therefore can compute $g^{ak} = (g^k)^a$ and recover *m*. An efficient discrete logarithm algorithm would enable a cryptanalyst to compute either *a* or *k*, and would therefore make this scheme insecure also.

T. ElGamal [26] has also proposed a novel signature scheme that uses exponentiation in fields $GF(p)$, p a prime. User A, who wishes to sign messages electronically, publishes a prime p, a primitive root g modulo p, and an integer y, $1 \le y \le p-1$, which is generated by choosing a random integer a, which is kept secret, and setting $y = g^a$. (The prime p and the primitive root g can be the same for all the users of the system, in which case only y is special to user A.) To sign a message m, $1 \le m \le p-1$, user A provides a pair of integers (r,s), $1 \le r,s \le p-1$, such that

$$g^m \equiv y^r \, r^s \pmod{p} \, . \tag{2.1}$$

To generate r and s, user A chooses a random integer k with $(k, p-1) = 1$ and computes

$$r = g^k \, .$$

Since $y = g^a$, this means that s has to satisfy

$$g^m \equiv g^{ar + ks} \pmod{p} \, , \tag{2.2}$$

which is equivalent to

$$m \equiv ar + ks \pmod{p-1} \, . \tag{2.3}$$

Since $(k, \; p-1) = 1$, there is a unique solution to (2.3) modulo $p-1$, and this solution is easy to find for user A, who knows a, r, and k. An efficient discrete logarithm algorithm would make this scheme insecure, since it would enable the cryptanalyst to compute a from y. No way has been found for breaking this scheme without the ability to compute discrete logarithms, and so the scheme appears quite attractive. It is not as fast as the Ong-Schnorr-Shamir signature scheme [50], but since several versions of that scheme were recently broken by Pollard, it should not be considered for use at the present time. The ElGamal scheme appears to be about as secure as the RSA scheme for moduli of the same length, as we will see later, although it does expand bandwidth, with the signature being twice as long as the message.

The presumed intractability of the discrete logarithm problem is crucial also for the Blum-Micali construction [9] of a cryptographically strong random number generator. What they show is that it is possible to compute a long sequence that is obtained deterministically from a short random sequence, and in which successive bits cannot be predicted efficiently from the preceding ones without the ability to compute discrete logarithms efficiently.

A scheme whose security is essentially equivalent to that of the Diffie - Hellman scheme was recently published by Odoni, Varadharajan, and Sanders [49]. These authors proposed taking a matrix B over $GF(p)$ which is the companion matrix of an irreducible polynomial $f(x)$ of degree m over $GF(p)$. The Diffie - Hellman scheme would then be implemented by replacing the primitive element g by the matrix B, so that pairs of users would transmit matrices B^a and B^b to each other, where a and b are the two random integers chosen by the two users. However, the matrix ring generated by B is isomorphic to the field $GF(p^m)$, so this scheme does not provide any additional security. The more sophisticated scheme proposed in [49], with the matrix B being obtained from several companion matrices of irreducible polynomials of degrees $m_1, \ldots, m_s$ can also be shown to be reducible to the problem of computing discrete logarithms in the fields $GF(p^{m_i})$ separately.

Finally, we mention that the ability to compute quantities generalizing discrete logarithms in rings of integers modulo composite integers would lead to efficient integer factorization algorithms [5,40,45,52].

3. Some special algorithms

In this section we discuss briefly some algorithms that apparently don't work very well and then we discuss a very useful algorithm that works well only when all the prime divisors of $q-1$ are of moderate size.

The first method we discuss was not designed as an algorithm at all. In a field $GF(p)$, p a prime, any function from the field to itself can be represented as a polynomial. Wells [64] has shown that for any u, $1 \le u \le p-1$, if g is a primitive root modulo p, then one can write

$$\log_g u \equiv \sum_{j=1}^{p-2} (1-g^j)^{-1} u^j \pmod{p} . \tag{3.1}$$

This formula is clearly useless computationally, but it is interesting that such an explicit form for the discrete logarithm function exists.

The Herlestam-Johannesson method [32] was designed to work over the fields $GF(2^n)$, and was reported by those authors to work efficiently for fields as large as $GF(2^{31})$. However, the heuristics used by those authors in arguing that the method ought to work efficiently in larger fields as well seem to be very questionable. As usual, $GF(2^n)$ is represented as polynomials over $GF(2)$ modulo some fixed irreducible polynomial $f(x)$ of degree n over $GF(2)$. In order to compute the logarithm of $h(x)$ to base x, Herlestam and Johannesson proposed to apply a combination of the transformations

$$h(x) \rightarrow h(x)^{2^r} ,$$

$$h(x) \rightarrow x^{-2^s} h(x)$$

so as to minimize the Hamming weight of the resulting polynomial, and apply this procedure iteratively until an element of low weight, for which the logarithm was known, was reached. There is no reason to expect such a strategy to work, and considerable numerical evidence has been collected which shows that this method is not efficient [13,67], and is not much better than a random walk through the field. However, some unusual phenomena related to the algorithm have been found whose significance is not yet understood [13,57]. In particular, the algorithm does not always behave like a random walk, and its performance appears to depend on the choice of the polynomial defining the field. These observations may be due to the small size of the fields that were investigated, in which case their significance would be slight.

Another approach to computing discrete logarithms in fields $GF(2^n)$ was taken by Arazi [3]. He noted that if one can determine the parity of the discrete logarithm of u, then one can quickly determine the discrete logarithm itself. Arazi showed that one can determine the parity of discrete logarithms to base g fast if g satisfies some rather complicated conditions. Since being able to compute discrete logarithms to one base enables one to compute them to any other base about equally fast (as will be discussed in Section 5), it would suffice to find any g that satisfies Arazi's condition. However, so far no algorithm has been found for finding such primitive elements g in large fields $GF(2^n)$, nor even a proof that any such elements exist. It was shown by this author that primitive elements g satisfying another set of conditions originally proposed by Arazi, which were more stringent than those of [3], do exist in fields $GF(2^n)$ for $2 \leq n \leq 5$, but not for $6 \leq n \leq 9$. Thus while the ideas of [3] are interesting and may be useful in future work, they appear to be of little practical utility at this moment.

We next discuss a very important algorithm that was published by Pohlig and Hellman [51], and whose earlier independent discovery they credit to Roland Silver. This algorithm computes discrete logarithms over $GF(q)$ using on the order of $\sqrt{p}$ operations and a comparable amount of storage, where p is the largest prime factor of $q-1$. In fact, there is a time-memory tradeoff that can be exploited, and Pohlig and Hellman [51] showed that if

$$q-1 = \prod_{i=1}^{k} p_i^{n_i} , \tag{3.2}$$

where the p_i are distinct primes, and if $r_1, \ldots, r_k$ are any real numbers with $0 \le r_i \le 1$, then logarithms over $GF(q)$ can be computed in

$$O\left(\sum_{i=1}^{k} n_i (\log q + p_i^{1-r_i} (1 + \log p_i^{r_i}))\right)$$

field operations, using

$$O\left(\log q \sum_{i=1}^{k} (1 + p_i^{r_i})\right)$$

bits of memory, provided that a precomputation requiring

$$O\left(\sum_{i=1}^{k} (p_i^{r_i} \log p_i^{r_i} + \log q)\right)$$

field operations is carried out first.

We now present a sketch of the above algorithm. Suppose that g is some primitive element of $GF(q)$, $x \in GF(q) - \{0\}$, and we wish to find an integer a, $1 \le a \le q-1$, such that

$$x = g^a . \tag{3.3}$$

Because of the Chinese Remainder Theorem, we only need to determine a modulo each of the $p_i^{n_i}$. Suppose that $p = p_i$ and $n = n_i$ for some i. Let

$$a \equiv \sum_{j=0}^{n-1} b_j p^j \pmod{p^n} .$$

To determine b_0, we raise x to the $(q-1)/p$ power:

$$y = x^{\frac{q-1}{p}} = g^{a \frac{q-1}{p}} = (q^{\frac{q-1}{p}})^{b_0} ,$$

and note that y is one of only p elements, namely

$$h^0 = 1, h^1, h^2, \ldots, h^{p-1} ,$$

where

$$h = g^{(q-1)/p} .$$

How one determines b_0 we will describe below. Once we have determined b_0, we can go on to determine b_1 by forming

$$(xg^{-b_0})^{(q-1)/p^2} = h^{b_1} ,$$

and so one.

The value of b_0 is determined using Shanks' "baby steps-giant steps" technique. We are given y, and we need to find m such that $y = h^m$, $0 \le m \le p-1$. If $r \in \mathbf{R}$ is given, $0 \le r \le 1$, we form

$$u = \lceil p^r \rceil .$$

Then there exist integers c and d such that

$$m = cu + d, \quad 0 \le d \le u-1, \quad 0 \le c < p/u .$$

Hence finding m is equivalent to finding integers c and d in the above ranges which satisfy

$$h^d \equiv yh^{-cu} .$$

To find such c and d, we can precompute h^d for $0 \le d \le n-1$ and then sort the resulting values. We then compute yh^{-cu} for $c = 0, 1, \ldots$, and check each value for a match with the sorted table of values of y^d. The precomputation and sorting take $O(p^2 \log p)$ operations (note that these steps have to be done only once for any given field), and there are $O(p^{1-r})$ values of yh^{-cu} to be computed.

The Silver-Pohlig-Hellman algorithm is efficient whenever all the prime factors of $q-1$ are reasonably small. (It is most efficient in fields in which q is a Fermat prime, $q = 2^m + 1$, for which there is another polynomial-time discrete logarithm method [41].) Therefore great care has to be taken in selecting the fields $GF(q)$ for use in cryptography. This question will be discussed further in Section 8.

We conclude this section by mentioning two interesting randomized algorithms due to Pollard [52]. One of them computes discrete logarithms in fields $GF(q)$ in time roughly $q^{1/2}$. The other algorithm finds the discrete logarithm of an element in time roughly $w^{1/2}$, if that logarithm is known to lie in an interval of size w.

4. A subexponential discrete logarithm method

This section presents the fastest known general purpose discrete logarithm method. The basic ideas are due to Western and Miller [65] (see also [47]). The algorithm was invented independently by Adleman [1], Merkle [46], and Pollard [52], and its computational complexity was partially analyzed by Adleman [1]. We will refer to it as the index-calculus algorithm. Previous authors were concerned largely with the fields $GF(p)$, where p is a prime. Here the method will be presented as it applies to the fields $GF(2^n)$, since they are of greatest cryptographic interest. An extensive asymptotic analysis of the running time of the algorithm in this and the related cases $GF(p^n)$ with p fixed and $n \rightarrow \infty$ was given recently by Hellman and Reyneri [30]. As will be shown below, their estimates substantially overestimate the running time of this algorithm.

Recently some improvements on the index-calculus method as it applies to the fields $GF(2^n)$ were made by I. Blake, R. Fuji-Hara, R. Mullin, and S. Vanstone [8] which make it much more efficient, although these improvements do not affect the asymptotics of the running time. Even more recently, D. Coppersmith [18,19] has come up with a dramatic improvement on the $GF(2^n)$ version of the algorithm (and more generally on the $GF(p^n)$ version with p fixed and $n \rightarrow \infty$) which is much faster and even has different asymptotic behavior. More recently, a whole series of improvements on the basic algorithm have been discovered [20]. They do not approach the Coppersmith algorithm in asymptotic performance, but they do apply to fields $GF(p)$ as well as $GF(2^n)$ and they can be used to motivate Coppersmith's algorithm (although they did not perform this function, having come afterwards), so we briefly sketch them as well.

The model of computation we will assume in this section is that of the Random Access Machine (RAM), with no parallel computation. In Section 6 we will discuss what effect lifting this restriction might have. The index-calculus algorithm, at least in the form presented here is a probabilistic method in that the analysis of its running time relies on assumptions about randomness and independence of various polynomials which seem reasonable but at present cannot be proved.

Before presenting the algorithm, it is necessary to specify the notation that will be used. As usual, we regard the field $GF(2^n)$ as the ring of polynomials over $GF(2)$ modulo some irreducible polynomial $f(x)$ of degree n. Hence all elements $g \in GF(2^n)$ can be regarded as polynomials $g(x)$ over $GF(2)$ of degree $< n$.

One very important factor in analyzing the performance of the index-calculus algorithm over $GF(2^n)$ is that polynomials over $GF(2)$ are very easy to factor. Algorithms are known [7,16,36,55] that can factor $g(x)$ in time polynomial in the degree of $g(x)$. Since the running time of the index-calculus algorithm in $GF(2^n)$ is much higher (of the form $\exp(c(n \log n)^{1/2})$ for the basic version and of the form $\exp(c' n^{1/3} (\log n)^{2/3})$ for the Coppersmith version), we will neglect the time needed to factor polynomials in this section, since we will be concerned here with asymptotic estimates. In Section 6 we will perform a more careful analysis for some specific values of n.

Suppose that $g(x)$, a polynomial of degree $< n$ over $GF(2)$, is a primitive element of $GF(2^n)$. The index-calculus method for computing discrete logarithms in $GF(2^n)$ with respect to the base $g(x)$ consists of two stages. The first stage, which is by far the more time and space consuming, consists of the construction of a large data base. This stage only has to be carried out once for any given field. The second stage consists of the computation of the desired discrete logarithms.

We now present the basic version of the index-calculus algorithm. The initial preprocessing stage, which will be described later, consists of the computation of the discrete logarithms (with respect to $g(x)$) of a set S of chosen elements of $GF(2^n)$. The set S usually consists of all or almost all the irreducible polynomials over $GF(2)$ of degrees $\leq m$, where m is appropriately chosen. Once the preprocessing stage is completed, logarithms can be computed relatively rapidly. The basic idea is that given $h = h(x)$, to find $a \in Z^+$ such that

$$h \equiv g^a \pmod{f} ,$$

one chooses a random integer s, $1 \leq s \leq 2^n - 1$, and computes

$$h^* \equiv h\, g^s \pmod{f} , \quad \deg h^* < n . \tag{4.1}$$

The reduced polynomial h^* is then factored into irreducible polynomials and if all its factors are elements of S, so that

$$h^* \equiv h\ g^s \equiv \prod_{v \in S} v^{b_v(h^*)} \pmod{f} , \tag{4.2}$$

then

$$\log_g h \equiv \sum_{v \in S} b_v(h^*) \log_g v - s \pmod{2^n - 1} . \tag{4.3}$$

In the form in which we have presented it so far, it is possible to obtain a fully rigorous bound for the running time of the second stage. The polynomials h^* in (4.1) behave like random polynomials over $GF(2)$ of degree $< n$. Let $p(k, m)$ denote the probability that a polynomial over $GF(2)$ of degree exactly k has all its irreducible factors of degrees $\leq m$; i.e., if $N(k,m)$ is the number of polynomials $w(x) \in GF(2)[x]$ such that $\deg w(x) = k$ and

$$w(x) = \prod_i u_i(x)^{c_i} , \quad \deg u_i(x) \leq m ,$$

then

$$p(k,m) = \frac{N(k,m)}{N(k,k)} = \frac{N(k,m)}{2^k} . \tag{4.4}$$

We expect that if S does consist of the irreducible polynomials of degrees $\leq m$, the reduced polynomial h^* in (4.1) will factor as in (4.2) with probability approximately $p(n,m)$, and that approximately $p(n,m)^{-1}$ of the polynomials of the form (4.1) will have to be generated before the second stage of the algorithm succeeds in finding the discrete logarithm of $h(x)$. (This reasoning explains why the set S is usually chosen to consist of all irreducible polynomials of degrees $\leq m$ for some fixed m; any other set of polynomials of equal cardinality is expected to have a smaller chance of producing a factorization of the form (4.2).)

The function $p(n,m)$ can be evaluated fairly easily both numerically and asymptotically. Appendix A presents the basic recurrences satisfied by $N(n,m)$ (from which $p(n,m)$ follows immediately by (4.4)), and shows that as $n \to \infty$ and $m \to \infty$ in such a way that $n^{1/100} \leq m \leq n^{99/100}$, (which is the range of greatest interest in the index calculus algorithm),

$$p(n,m) = \exp((1 + o(1)) \frac{n}{m} \log_e \frac{m}{n}) . \tag{4.5}$$

Appendix B consists of a table of $p(n, m)$ for a selection of values of n and m, which was computed using the recurrences in Appendix A. Approximations better than that of (4.5) for $p(n,m)$ can be obtained with more work, but for practical purposes the table of Appendix B is likely to be quite adequate and is more accurate to boot. The analysis of Hellman and Reyneri [30] relied on an estimate of $p(n, m)$ that was essentially equivalent to

$$p(n, m) \geq \exp((1+o(1)) \frac{n}{m} \log_e \frac{2e}{n}) ,$$

which while true, is much weaker than (4.5).

The polynomials h^* are always of degree $\leq n-1$, and have degree $n-k$ with probability 2^{-k}. Hence the probability that h^* factors in the form (4.2) is better approximated by

$$\sum_{k=1}^{n} 2^{-k} p(n-k, m) ,$$

which is approximately

$$p(n, m) \frac{(ne/m)^{1/m}}{2-(ne/m)^{1/m}} ,$$

as follows from the results of Appendix A. The last quantity above is $\sim p(n, m)$ as $n \to \infty$, $n^{1/100} \leq m \leq n^{99/100}$. Hence asymptotically this effect is unimportant, although for small values of n and m it can make a difference; for example, for $n = 127$ and $m = 17$ we obtain $1.51 p(127, 17)$ as the correct estimate of the probability that h^* will factor in the form (4.2).

The relation (4.5) shows that the expected running time of the second stage of the algorithm, as it has been presented so far, is approximately

$$p(n,m)^{-1} = (\frac{n}{m})^{(1+o(1)) n/m} . \tag{4.6}$$

It was recently observed by Blake, Fuji-Hara, Mullin, and Vanstone [8] that this stage can be speeded up very substantially, although at the cost of not being able to provide an equally rigorous bound for the running time. Their idea is not to factor the polynomial h^* defined by (4.1) directly, but instead to find two polynomials w_1 and w_2 such that

$$h^* \equiv \frac{w_1}{w_2} \pmod{f} , \tag{4.7}$$

and such that deg $w_i \lesssim n/2$ for $i = 1,2$. Once that is done, the w_i are factored, and if each is divisible only by irreducibles from *S*, say

$$w_i = \prod_{v \in S} v^{c_v(i)} , \tag{4.8}$$

then

$$\log_g h \equiv \sum_{v \in S} (c_v(1) - c_v(2)) \log_g v - s \pmod{2^n - 1} . \tag{4.9}$$

The advantage of this approach is that if the w_i behave like independently chosen random polynomials of degree ~ $n/2$, as seems reasonable, then the probability that both will factor into irreducibles of degrees $\leq m$ is approximately $p([n/2], m)^2$, and therefore the expected number of polynomials h^* that have to be tested is on the order of

$$p([n/2], m)^{-2} = (\frac{n}{2m})^{(1+o(1))n/m} . \tag{4.10}$$

This is smaller than the quality in (4.8) by a factor of approximately $2^{n/m}$, and so is very important, provided the w_i can be generated fast.

The polynomials w_i can be generated very rapidly (in time polynomial in *n*) by applying the extended Euclidean algorithm [36,42] to h^* and *f*. This algorithm produces polynomials α and β over $GF(2)$ such that $\alpha h^* + \beta f = 1$, the greatest common divisor of h^* and *f*, and such that deg $\alpha <$ deg $f = n$, deg $\beta <$ deg $h^* < n$. To do this, the algorithm actually computes a sequence of triples of polynomials $(\alpha_j, \beta_j, \gamma_j)$ such that

$$\alpha_j h^* + \beta_j f = \gamma_j , \tag{4.11}$$

where the final $(\alpha_j, \beta_j, \gamma_j) = (\alpha, \beta, 1)$, deg $\gamma_1 >$ deg $\gamma_2 >$ ..., and where deg $\alpha_j \leq n - 1 -$ deg γ_j. If we choose that *j* for which deg γ_j is closest to $n/2$, then $w_1 = \gamma_j$ and $w_2 = \alpha_j$ will satisfy the congruence (4.7), and their degrees will be relatively close to $n/2$ most of the time. These w_1 and w_2 are not completely independent (for example, they have to be relatively prime), but on the other hand their degrees will often be less than $n/2$, so on balance it is not unreasonable to expect that the probability of both having a factorization of the form (4.8) should be close to $p([n/2],\ m)^2$.

The above observations justify the claim that the second stage of the index-calculus algorithm, as modified by Blake et al., ought to take on the order of $p([n/2],m)^{-2}$ operations on polynomials of degree $\leq n$ over $GF(2)$, where each such polynomial operation might involve on the order of n^3 bit operations. For small values of n, $p([n/2],m)$ can be found in Appendix B, while for very large n, the quantity on the right side of (4.10) ought to be a reasonable approximation to the running time of the second stage.

It is clear that the running time of the second stage can be decreased by increasing m. Doing that, however, increases both storage requirements and the running time of the first (preprocessing) stage of the algorithm. It is well known (see Appendix A) that the number of irreducible polynomials of degree $\leq m$ is very close to $m^{-1}2^{m+1}$, and for each one it is necessary to store roughly n bits, namely its logarithm (which is in the range $[1,2^n-1]$). This already puts a limit on how large m can be, but this limit is not very stringent, since these discrete logarithms can be stored on slow storage devices, such as tape. This is due to the fact that they are needed only once in the computation of each discrete logarithm by stage two, when both of the polynomials w_i are discovered to have factorizations of the form (4.8). Thus this argument does not exclude the use of values of m on the order of 40.

A much more severe limitation on the size of m and n is placed by the preprocessing first stage, which we now discuss. The basic idea there is to choose a random integer s, $1 \leq s \leq 2^n-1$, form the polynomial

$$h^* \equiv g^s \pmod{f}, \quad \deg h^* < n \ ,$$

and check whether h^* factors into irreducible factors from S. If it does, say

$$h^* = \prod_{v \in S} v^{b_v(h^*)} \ , \tag{4.12}$$

then we obtain the congruence

$$s \equiv \sum_{v \in S} b_v(h^*) \log_g v \pmod{2^n-1} \ . \tag{4.13}$$

Once we obtain slightly more than $|S|$ such congruences, we expect that they will determine the $\log_g v$, $v \in S$, uniquely modulo 2^n-1, and the first stage will be completed. There is a complication here in that

$2^n - 1$ is not in general a prime, so that solving the system (4.13) might require working separately modulo the different prime power divisors of $2^n - 1$ and using the Chinese Remainder Theorem to reconstruct the values of $\log_g v$. This complication is not very serious, and if it does occur, it should lead to a speedup in the performance of the algorithm, since arithmetic would have to be done on smaller numbers. In any case this complication does not arise when $2^n - 1$ is a prime. A general linear system of the form (4.13) for the $\log_g v$ takes on the order of $|S|^3$ steps to solve if we use straightforward gaussian elimination. (We neglect here multiplicative factors on the order of $O(n^2)$.) This can be lowered to $|S|^r$ for $r = 2.495548...$ using known fast matrix multiplication algorithms [21], but those are not practical for reasonably sized $|S|$. The use of Strassen's matrix multiplication algorithm [10] might be practical for large n, and would lower the running time to about $|S|^r$ with $r = \log_2 7 = 2.807...$. However, the systems of linear equations that arise in the index-calculus algorithms are quite special in that they are quite sparse. (i.e., there are only a few nonzero coefficients). It was only recently discovered that this sparseness can be effectively exploited, and systems (4.13) can be solved in time essentially $|S|^2$. This development will be described in Section 5.7.

Generation of $|S|$ congruences of the form (4.13) takes about

$$|S| \ p(n,m)^{-1}$$

steps if we use the algorithm as described above. If instead we use the Blake et al. modification described in connection with the second stage, in which instead of factoring h^* right away, we first express it in the form (4.7) with deg $w_i \lesssim n/2$, $i = 1,2$, and then factor the w_i, then generation of $|S|$ of the congruences (4.13) ought to take on the order of

$$|S| \ p([n/2],\ m)^{-2} \tag{4.14}$$

steps, where each step takes a polynomial number (in n) of bit operations. Thus the first stage of the algorithm takes on the order of

$$|S| \ p([n/2],\ m)^{-2} + |S|^2 \tag{4.15}$$

steps. Hence using our approximations to $p(k,\ m)$ and $|S|$ and discarding polynomial factors yields an estimate of the running time of the form

$$2^m \left(\frac{n}{2m}\right)^{n/m} + 2^{2m} \ . \tag{4.16}$$

(To be precise, the exponents in (4.16) should be multiplied by $1 + o(1)$.) The quantity

$$2^m \left(\frac{n}{2m}\right)^{n/m}$$

is minimized approximately for $m \sim c_1 (n \ \log_e n)^{1/2}$, where

$$c_1 = (2 \ \log_e 2)^{-1/2} = 0.8493... \ ,$$

in which case

$$2^m \left(\frac{n}{2m}\right)^{n/m} = \exp \ ((c_2 + o(1)) \ \sqrt{n \ \log_e \ n}) \quad \text{as} \quad n \to \infty \ , \tag{4.17}$$

where

$$c_2 = c_1 \ \log_2 \ 2 + (2c_1)^{-1} = (2 \ \log_e \ 2)^{1/2} = 1.1774... \ .$$

For $m \sim c_1 (n \ \log_e n)^{1/2}$, 2^{2m} is also of the form (4.17), so the time to solve the system of linear equations is of the same asymptotic form as the time needed to generate them.

If we modify the notation used by Pomerance [53] in his survey of integer factorization and let $M = M(n)$ represent any quantity satisfying

$$M = \exp((1 + o(1)) \ (n \ \log_e \ n)^{1/2}) \quad \text{as} \quad n \to \infty \ ,$$

then our analysis shows that the first stage of the basic index-calculus algorithm can be carried out in time $M^{1.178}$.

The time required by the second stage of the index-calculus algorithm to compute a single discrete logarithm is

$$M^{(2c_1)^{-1}} = M^{0.588...} \ .$$

This running time estimate is much lower than for the first stage. The space requirements of the second stage are essentially negligible. It is necessary to have access to the logarithms of the elements of S, which requires

$$\exp\ ((c_1 \log_e 2 + o(1))\ (n \log_e n)^{1/2})$$

bits of storage, but these logarithms are needed only once, and so they can be stored on a cheap slow-access device, such as tape.

Our estimates for the running time of the basic index-calculus algorithm for the fields $GF(2^n)$ are substantially smaller than those of Hellman and Reyneri [30]. This is due primarily to our use of a more accurate estimate for $p(n, m)$. The Blake et al. innovation which replaces the polynomial h^* by the quotient of two polynomials, each of roughly half the degree of h^* turns out not to affect the asymptotic estimate, since it improves the running time only by the factor $2^{n/m}$, which is $M^{o(1)}$ for $m \sim c(n \log_e n)^{1/2}$. However, for values of n that might be of practical interest, say $200 \le n \le 1000$, and best possible choices of m, this factor $2^{n/m}$ is very important, speeding up the algorithm by between two and ten orders of magnitude.

We next describe several algorithms that improve on the asymptotic performance of the basic index-calculus algorithm to an extent greater than the Blake et al. [8] modification. They are nowhere near as fast as the Coppersmith version, since they still run in time M^c for some constant $c > 0$, but they have the property that $c < c_2$. They are presented here very briefly in order to show the variety of methods that are available, and also to motivate the Coppersmith algorithm. Like the Coppersmith method, these variants depend on the polynomial $f(x)$ that defines the field being of a somewhat special form, namely

$$f(x) = x^n + f_1(x)\ , \tag{4.18}$$

where the degree of $f_1(x)$ is small. Since approximately one polynomial of degree n out of n is irreducible (cf. Appendix A), we can expect to find $f(x)$ of the form (4.18) with $\deg f_1(x) \lesssim \log_2 n$. (The $f_1(x)$ of smallest degrees for which $x^n + f_1(x)$ is irreducible for some interesting values of n are $f_1(x) = x+1$ for $n = 127$, $f_1(x) = x^9+x^6+x^5+x^3+x+1$ for $n = 521$, $f_1(x) = x^9+x^7+x^6+x^3+x+1$ for $n = 607$, and $f_1(x) = x^{11}+x^9+x^8+x^5+x^3+x^2+x+1$ for $n = 1279$.) As is explained in Section 5.2, this is not a severe restriction, since being able to compute logarithms rapidly in one representation of a field enables one to compute logarithms in any other representation just about as fast.

The first algorithm we discuss is one of several that have the same asymptotic performance. (The other algorithms in this group are described in [20], at least in the form applicable to fields $GF(p)$.) It is basically an adaptation of the Schroeppel factorization algorithm [20,53]. We assume that $f(x)$ is of the form (4.18) with deg $f(x) \le n/2$, say. This time we let $S = S_1 \cup S_2$, where S_1 consists of the irreducible polynomials of degrees $\le m$, and S_2 of polynomials of the form

$$x^k + g(x) , \quad \deg\ g(x) \le m , \tag{4.19}$$

where $k = \lceil n/2 \rceil$ is the least integer $\ge n/2$. Consider any $h_1(x),\ h_2(x)\ \varepsilon\ S_2$. If

$$h_i(x) = x^k + \tilde{h}_i(x) , \quad i = 1,2 ,$$

then, if we write $2k = n+a$, $a = 0$ or 1, we have

$$\begin{aligned} h_1(x)\ h_2(x) &= x^{2k} + x^k(\tilde{h}_1(x) + \tilde{h}_2(x)) + \tilde{h}_1(x)\tilde{h}_2(x) \\ &= x^a(f(x) + f_1(x)) + x^k(\tilde{h}_1(x)+\tilde{h}_2(x)) + \tilde{h}_1(x)\tilde{h}_2(x) \\ &\equiv x^k(\tilde{h}_1(x)+\tilde{h}_2(x)) + \tilde{h}_1(x)\tilde{h}_2(x) + x^a f_1(x)\ (\mathrm{mod}\ f(x)) , \end{aligned} \tag{4.20}$$

and so the polynomial on the right side of (4.20) is of degree roughly $n/2$ (for $m = o(n)$, as will be the case). If that polynomial, call it $h^*(x)$, factors into irreducible polynomials of degrees $\le m$, say

$$h^*(x) = \prod_{v \varepsilon S_1} v(x)^{b_v(h^*)} ,$$

then (4.20) yields a linear equation for the logarithms of the elements of S:

$$\log_g\ h_1 + \log_g\ h_2 \equiv \sum_{v \varepsilon S_1} b_v(h^*)\ \log_g\ v \quad (\mathrm{mod}\ 2^n - 1) . \tag{4.21}$$

Since each of S_1 and S_2 has on the order of 2^m elements, once we obtain about 2^m equation of the form (4.21), we ought to be able to solve them and obtain the discrete logarithms of the elements of S_1, which is what is desired. Now there are approximately 2^{2m} different pairs h_1, h_2 that can be tested, and if the h^* behave like random polynomials of degrees about $n/2$, each will factor into irreducibles of degrees $\le m$ with probability approximately $p([n/2],\ m)$. Hence we will have to perform about 2^{2m} polynomial time factorizations and obtain about $2^{2m}\ p([n/2],\ m)$ equations of the form (4.21). Therefore we need

$$2^{2m}\ p([n/2],\ m) \gtrsim 2^m , \tag{4.22}$$

and the work we do is on the order of 2^{2m}, since the linear equations can also be solved in this much time.

To minimize the running time, we choose the smallest m for which (4.22) is satisfied, and a brief computation shows that the right choice is $m \sim c_3 (n \log_e n)^{1/2}$ as $n \to \infty$, with $c_3 = (4 \log_e 2)^{-1/2}$, so that the running time of the first stage of this algorithm is

$$M^{c_4} = M^{0.8325...} \quad , \quad c_4 = (\log_e 2)^{1/2} \ . \tag{4.23}$$

The improvement in the exponent of M is the running time estimate of the first stage from 1.177... in the basic algorithm to 0.832... in the version above was due to the fact that this time, in order to obtain a linear equation we only had to wait for a single polynomial of degree about $n/2$ to split into low degree irreducibles, instead of a single polynomial of degree n or two polynomials of degree $n/2$. In the next algorithm, we obtain a further improvement by reducing to the problem of a single polynomial of degree about $n/3$ splitting into low degree irreducibles. The method is an adaptation of the so-called ‘‘cubic sieve’’ for factoring integers, which was invented by J. Reyneri some years ago and rediscovered independently several times since then (see [20] for further details). This time we assume that $f(x)$ has the form (4.18) with $\deg f_1(x) \le n/3$. We set $k = \lceil n/3 \rceil$ and let $S = S_1 \cup S_2$ with S_1 consisting of the irreducible polynomials of degrees $\le m$ and S_2 of polynomials of the form $x^k + h(x)$, $\deg h(x) \le m$. We consider pairs $h_1(x)$ and $h_2(x)$ with each $h_i(x)$ of degree $\le m$, and let

$$h^*(x) \equiv (x^k + h_1(x)) \ (x^k + h_2(x)) \ (x^k + h_1(x) + h_2(x)) \pmod{f(x)} \ , \tag{4.24}$$

$0 \le \deg h^*(x) < n$. We then have

$$h^*(x) \equiv x^{3k} + x^k (h_1^2 + h_1 h_2 + h_2^2) + h_1 h_2 (h_1 + h_2) \pmod{f} \ , \tag{4.25}$$

and since

$$x^{3k} \equiv x^a f_1(x) \pmod{f(x)}$$

for some a, $0 \le a \le 2$, we find that $h^*(x)$ is of degree about $k \sim n/3$ if $m = o(n)$. If $h^*(x)$ is divisible only by irreducibles in S_1, we obtain a linear equation relating logarithms of three elements of S_2 to those of elements of S_1. There are about 2^{2m} pairs $h_1(x), h_2(x)$ to test, and so if the $h^*(x)$ behave like random polynomials of degrees $\sim n/3$, we expect to obtain about $2^{2m} \ p([n/3], m)$ equations. Since there are about 2^m elements of S, we therefore need to choose m so that

$$2^{2m} \ p([n/3],\ m) \geq 2^m \ . \tag{4.26}$$

The time to run the algorithm is (within polynomial factors of n) 2^{2m}, both to form and factor the polynomials $h^*(x)$, and to solve the system of linear equations. A simple computation shows that the smallest m that satisfies (4.26) has $m \sim c_5 (n \log_e n)^{1/2}$, where $c_5 = (6 \log_e 2)^{-1/2}$, and the running time of the first phase of this algorithm is

$$M^{c_6} = M^{0.6797\ldots} \ , \ \text{where } c_6 = (2(\log_e 2)/3)^{1/2} \ . \tag{4.27}$$

The running times of the second phases of the two algorithms presented above can be improved beyond what is obtained by using the strategy of the basic variant, but we will not discuss that subject. Details can be found in [20], in the case of fields $GF(p)$, p a prime, and it is easy to adapt those methods to the fields $GF(2^n)$.

The variants of the index-calculus algorithm presented above raise the question of whether they can be generalized so as to give even faster algorithms. The obvious idea is to use more than three factors and choose those factors in such a way that the product will reduce modulo $f(x)$ to a polynomial of low degree. A very clever way to do this was found by Coppersmith [18,19]. However, his work was motivated by different considerations.

We next present the Coppersmith variation [18,19] on the index-calculus algorithm. Unlike the basic algorithm, which runs in time roughly of the form $\exp(n^{1/2})$ in fields $GF(2^n)$, this new variation runs in time which is roughly of the form $\exp(n^{1/3})$. Unlike the basic version, though, the Coppersmith variant does not apply to the fields $GF(p)$ with p prime. Just like the algorithms presented above, the Coppersmith algorithm relies on several unproved assumptions. Since these assumptions are supported by both heuristic reasoning and empirical evidence, though, there seems to be no reason to doubt the validity of the algorithm.

The Coppersmith algorithm was inspired to a large extent by the Blake et al. [8] method of systematic equations, which is explained in Section 5.1, and which yields many linear equations involving logarithms at very low cost. Like the systematic equations method, it depends on the polynomial $f(x)$ being of a the special form (4.18) with $f_1(x)$ of very low degree.

We now discuss the first stage of the Coppersmith variant of the index-calculus algorithm. We assume that the field $GF(2^n)$ is defined by a polynomial $f(x)$ that is of the form (4.18) with deg $f_1(x) \lesssim \log_2 n$. The first stage consists again of the computation of logarithms of $v \in S$, where S consists of irreducible polynomials of degrees $\leq m$, but now m will be much smaller, on the order of $n^{1/3} (\log_e n)^{2/3}$. We will also assume that $g(x) \in S$, since it follows from Section 5.2 that this restriction does not affect the running time of the algorithm.

The essence of the Blake et al. [8] improvement of the basic index-calculus algorithm is that it replaced the factorization of a single polynomial of degree about n by the factorization of two polynomials of degrees about $n/2$ each. The essence of the two improvements discussed above was that they rely on the factorization of polynomials of degrees about $n/2$ and $n/3$, respectively, into low degree irreducibles. The essence of the Coppersmith [18,19] improvement is that it instead relies on factorization of two polynomials of degrees on the order of $n^{2/3}$ each. The lower the degree of the polynomials being factored, the greater the probability that they will consist only of small degree irreducible factors. To accomplish this lowering of the degree, take $k \in Z^+$ (k will be chosen later so that 2^k is on the order of $n^{1/3} (\log_e n)^{-1/3}$) and define

$$h = \lfloor n \, 2^{-k} \rfloor + 1 \; . \tag{4.28}$$

Pick $u_1(x)$ and $u_2(x)$ of degrees $\leq B$ (B will be chosen later to be on the order of $n^{1/3} (\log_e n)^{2/3}$) with $(u_1(x), u_2(x)) = 1$, and set

$$w_1(x) = u_1(x) x^h + u_2(x) \; . \tag{4.29}$$

Next let

$$w_2(x) \equiv w_1(x)^{2^k} \pmod{f(x)} \; , \quad \deg \, w_2(x) < n \; . \tag{4.30}$$

We then have

$$\begin{aligned} w_2(x) &\equiv u_1(x^{2^k}) x^{h 2^k} + u_2(x^{2^k}) \pmod{f(x)} \; , \\ &= u_1(x^{2^k}) x^{h 2^k - n} f_1(x) + u_2(x^{2^k}) \; . \end{aligned} \tag{4.31}$$

If B and 2^k are on the order of $n^{1/3}$, then h is on the order of $n^{2/3}$, $h \, 2^k - n$ is on the order of $n^{1/3}$, and so both $w_1(x)$ and $w_2(x)$ have degrees on the order of $n^{2/3}$. Since

$$\log_g \ w_2(x) \equiv 2^k \log_g \ w_1(x) \pmod{2^n - 1} ,$$

if both $w_1(x)$ and $w_2(x)$ have all their irreducible factors in S we obtain a linear equation for the $\log_g \ v$, $v \in S$. (The restriction $(u_1(x), u_2(x)) = 1$ serves to eliminate duplicate equations, since the pairs $u_1(x), u_2(x)$ and $u_1(x)t(x), u_2(x)t(x)$ produce the same equations.)

We next consider the Coppersmith algorithm in greater detail. We need to obtain about $|S|$ linear equations for the $\log_g \ v$, $v \in S$. Now

$$\deg \ w_1(x) \le B + h ,$$

$$\deg \ w_2(x) \le B \cdot 2^k + 2^k + \deg \ f_1(x) ,$$

so if $w_1(x)$ and $w_2(x)$ behave like independent random polynomials of those degrees, then the probability that both $w_1(x)$ and $w_2(x)$ have all their irreducible factors in S is approximately

$$p(B+h, \ m) p(B2^k + 2^k, \ m) \ . \tag{4.32}$$

Of course $w_1(x)$ and $w_2(x)$ are neither independent nor random. However, as far as their factorizations are concerned, it does not appear unreasonable to expect that they will behave like independent random polynomials, and this does turn out to hold in the case $n = 127$ studied by Coppersmith [18,19]. Therefore to obtain $|S| \sim m^{-1} 2^{m+1}$ equations we need to satisfy

$$2^{2B} \ p(B+h, \ m) p(B2^k + 2^k, \ m) \gtrsim 2^m \ . \tag{4.33}$$

The work involved consists of generating approximately 2^{2B} polynomials $w_1(x)$ and testing whether both $w_1(x)$ and $w_2(x)$ have all their irreducible factors in S. Once these roughly 2^m equations are generated, it becomes necessary to solve them, which takes about 2^{2m} operations. The estimate (4.5) shows that to minimize the running time, which is approximately

$$2^{2B} + 2^{2m} ,$$

subject to (4.33), it is necessary to take

$$2^k \sim \alpha \; n^{1/3} \; (\log_e \; n)^{-1/3} \;, \tag{4.34a}$$

$$m \sim \beta \; n^{1/3} \; (\log_e \; n)^{2/3} \;, \tag{4.34b}$$

$$B \sim \gamma \; n^{1/3} \; (\log_e \; n)^{2/3} \;, \tag{4.34c}$$

as $n \to \infty$, where α, β, and γ are bounded away from both zero and infinity. Under these conditions we find that the running time of the first stage of the algorithm is

$$K^{2\gamma \log_e 2} + K^{2\beta \log_e 2} \;, \tag{4.35}$$

where $K = K(n)$ denotes any quantity that satisfies

$$K = \exp\; ((1+o(1)) \; n^{1/3} \; (\log_e \; n)^{2/3}) \;, \tag{4.36}$$

and this is subject to the condition

$$2 \; \gamma \; \log_e \; 2 \; - \; \frac{1}{3\alpha\beta} \; - \; \frac{\alpha\gamma}{3\beta} \geq (1+o(1)))\beta \; \log_e \; 2 \;. \tag{4.37}$$

Let us now regard α, β, and γ as continuous variables. Since the estimate (4.35) does not depend on α, we can choose α freely. The quantity on the left side of (4.37) is maximized for

$$\alpha = \gamma^{-1/2} \;, \tag{4.38}$$

and for this choice of α, (4.37) reduces to (after neglecting the $1+o(1)$ factor)

$$2\gamma \; \log_e \; 2 \geq \beta \; \log_e \; 2 \; + \; \frac{2}{3} \; \beta^{-1}\gamma^{1/2} \;. \tag{4.39}$$

To minimize the asymptotic running time of the algorithm, we have to choose β and γ so that (4.39) is satisfied and $\max\,(2\gamma,\, 2\beta)$ is minimized. A short calculation shows that the optimal choice is obtained when $\gamma = \beta$ and equality holds in (4.37), which yields

$$\beta = 2^{2/3}3^{-2/3} \;\; (\log_e \; 2)^{-2/3} = 0.9743... \;. \tag{4.40}$$

The running time for this choice is

$$K^{2\beta \log_e 2} = K^{1.3507...} \;,$$

and the space required is $K^{\beta \log_e 2} = K^{0.6753...}$.

The analysis above assumed that α, β, and γ could all be treated as continuous variables. This is essentially true in the case of β and γ, but not in the case of α, since (4.34a) has to hold with k a positive integer. Since the analysis is straightforward but tedious, we do not discuss the general situation in detail but only mention that the running time of the Coppersmith algorithm and the space required are of the form K^u, where u is a function of $\log_2 (n^{1/3} (\log_e n)^{2/3})$ which is periodic with period 1. The minimal value of u is $2\beta \log_e 2$, with β given by (4.40), while the maximal value of u is $3^{2/3}\beta \log_e 2 = (2.08008...) \beta \log_e 2$. Thus we are faced with the not uncommon situation in which the running time of the algorithm does not satisfy a simple asymptotic relation but exhibits periodic oscillations.

We next discuss the second stage of the Coppersmith algorithm, which computes logarithms of arbitrary elements. It is somewhat more involved than the second stage of the basic version of the algorithm. If h is a polynomial whose logarithm is to be determined, then Coppersmith's second stage consists of a sequence of steps which replace h by a sequence of polynomials of decreasing degrees. The first step is similar to the second stage of the basic algorithm and consists of selecting a random integer s, forming h^* as in (4.1), and checking whether h^* has all its irreducible factors of degrees $\leq n^{2/3} (\log_e n)^{1/3}$, say. (In practice, one would again replace h^* by w_1/w_2, where the degrees of the w_i are $\lesssim h/2$, and the bound on the degrees of the irreducible factors might be somewhat different, but that is not very important.) The probability of success is approximately $p(n,\ n^{2/3}(\log_e n)^{1/3})$, so we expect to succeed after

$$p(n,\ n^{2/3}(\log_e n)^{1/3})^{-1} = K^{\log_e 3} = K^{1.098...} \tag{4.41}$$

trials. When we do succeed with some value of s, we obtain

$$h \equiv g^{-s} \prod_i u_i \pmod{f(x)} ,$$

where the u_i are of degrees $\leq n^{2/3} (\log n)^{1/3}$, and there are $< n$ of them (since their product is a polynomial of degree $< n$). This then yields

$$\log_g h \equiv -s + \sum_i \log_g u_i \pmod{2^n - 1} , \tag{4.42}$$

and so if we find the $\log_g u_i$, we obtain $\log_g h$.

Suppose next that u is a polynomial of degree $\leq B \leq n^{2/3} (\log n)^{1/3}$ (say one of the u_i above, in which case $B = n^{2/3} (\log n)^{1/3}$). We again reduce the problem of computing $\log_g u$ to that of computing logarithms of several polynomials of lower degrees. We select 2^k to be a power of 2 close to $(n/B)^{1/2}$ (precise choice to be specified later), and let

$$d = \lfloor n 2^{-k} \rfloor + 1 \ . \tag{4.43}$$

Consider polynomials

$$w_1(x) = v_1(x) x^d + v_2(x) \ , \tag{4.44}$$

where deg $v_1(x)$, deg $v_2(x) \leq b$ (b to be specified later), $(v_1(x), v_2(x)) = 1$, and $u(x) | w_1(x)$. If

$$w_2(x) \equiv w_1(x)^{2^k} \pmod{f(x)} \ , \quad \deg\ w_2(x) < n \ , \tag{4.45}$$

then (for b small)

$$w_2(x) = v_1(x^{2^k}) x^{d2^k - n} f_1(x) + v_2(x^{2^k}) \ ,$$

and thus $w_1(x)$ and $w_2(x)$ both have low degrees. If $w_1(x)/u(x)$ and $w_2(x)$ both factor into irreducible polynomials of low degree, say

$$w_1(x) = u(x) \prod_i s_i(x) \ ,$$

$$w_2(x) = \prod_j t_j(x) \ ,$$

then we obtain

$$\sum_j \log_g t_j(x) \equiv \log_g w_2(x) \equiv 2^k \log_g w_1(k)$$

$$\equiv 2^k (\log_g u(x) + \sum_i \log_g s_i(x)) \pmod{2^n - 1} \ .$$

This reduces the computation of $\log_g u$ to the computation of the $\log_g t_j$ and the $\log_g u_i$. We next analyze how much of a reduction this is. The probability that $w_1(x)/u(x)$ and $w_2(x)$ both factor into irreducible polynomials of degrees $\leq M$ is approximately

$$p(d + b - \deg\ u(x), M)\ p(b 2^k + 2^k + \deg\ f_1(x), M) \ ,$$

and the number of pairs of polynomials $v_1(x)$, $v_2(x)$ of degrees $\leq b$ with $(v_1(x), v_2(x)) = 1$ and

$u(x) \mid w_1(x)$ is approximately

$$2^{2b - \deg\ u(x)}\ .$$

(Divisibility by $u(x)$ is determined by a set of deg $u(x)$ linear equations for the coefficients of $v_1(x)$ and $v_2(x)$.) Hence to find $v_1(x)$ and $v_2(x)$ such that $w_1(x)$ and $w_2(x)$ factor in the desired fashion we select b to be approximately

$$(n^{1/3}\ (\log_e\ n)^{2/3} (\log_e\ 2)^{-1}\ +\ \deg\ u(x))/2\ , \tag{4.46}$$

and select 2^k to be the power of 2 nearest to $(n/b)^{1/2}$. We then expect to obtain the desired factorization in time

$$K\ =\ \exp((1+o(1))\, n^{1/3} (\log_e\ n)^{2/3})\ ,$$

with M being the largest integer for which

$$Kp(d+b-\deg\ u(x),\ M)\, p(b2^k+2^k+\deg\ f_1(x),\ M)\ \geq\ 1\ . \tag{4.47}$$

If $B \sim n^{2/3} (\log_e\ n)^{1/3}$ (as occurs in the first step of the second stage of the Coppersmith algorithm), we find that we can take $M \sim cn^{1/2} (\log_e\ n)^{3/2}$, and if $B \sim cn^{1/2} (\log_e\ n)^{3/2}$, then we can take $M \sim c' n^{5/12} (\log_e\ n)^{25/12}$. More generally, it is also easy to show that if $B \geq n^{1/3} (\log_e\ n)^{2/3}$, say, then we can take $M \leq B/1.1$, so that each iteration decreases the degrees of the polynomials whose logarithms we need to compute by a factor ≥ 1.1, while raising the number of these polynomials by a factor $\leq n$. When $B \leq (1.1)^{-1} n^{1/3} (\log_e\ n)^{2/3}$, the polynomial $u(x)$ is already in our data base, and we only need to read off its logarithm. Thus we expect to perform

$$\leq\ \exp(c''(\log\ n)^2)\ =\ K^{o(1)}$$

iterations of this process, each iteration taking K steps.

We have shown that the second stage of the Coppersmith algorithm can compute individual logarithms in time $K^{1.098...}$. In fact, with slightly more care the exponent of K can be lowered substantially. We do not do it here, since the main point we wish to make is that as in the basic algorithm, the second stage of the Coppersmith variant requires very little time and negligible space, compared to the first stage.

This section was devoted almost exclusively to the asymptotic analysis of the index-calculus algorithms on a random access machine. In Section 6 we will consider the question of estimating the running time of this algorithm for some concrete values of n, including the possible effects of the use of parallel processors. In the next section we will discuss several variations on the algorithm as it has been presented so far.

5. Further modifications of the index-calculus algorithm

Section 4 was concerned largely with the asymptotic behavior of the index-calculus algorithm in fields $GF(2^n)$. This section will discuss several technical issues related to both the basic algorithm and the Coppersmith version. The most important of them is that of efficient solutions to systems of linear equations, discussed in Section 5.7. The fact that the equations that occur in index-calculus algorithms can be solved fast is a recent discovery which affects the estimates of the running time both asymptotically and in practice.

This section also presents a variety of modifications of both the basic algorithm and of the Coppersmith version, which do not affect the asymptotics of the running times very much, but which are very important in practice. The most significant of these variations is that of Section 5.6. That variation speeds up the first phase of the Coppersmith algorithm by two or three orders of magnitude in fields that might be of practical interest. The variations presented here are not analyzed in exhaustive detail because their exact contributions depend on the hardware and software in which the algorithm is implemented. The purpose here is to obtain rough estimates of the performance of the algorithm with the best currently conceivable techniques. These estimates will be used in the next section to evaluate how large n ought to be to offer a given level of security.

5.1 Systematic equations

The first stage of the index-calculus algorithm involves the collection of slightly over $|S|$ linear equations for the logarithms of the polynomials $v \in S$ and then the solution of these equations. The reason the Coppersmith version is so much faster than the Blake et al. version is that by dealing with pairs of polynomials of degree around $n^{2/3}$ as opposed of degree about $n/2$, it increases the probability of finding an additional equation for the $\log_g v$, $v \in S$. In fact, for the fields $GF(2^n)$, Blake et al. had some methods for obtaining large numbers of equations at very low cost per equation. They called the equations obtained this way ''systematic.'' They were able to obtain upwards of one half of the required number of equations that way, but never all. Their methods in fact inspired Coppersmith to invent his version of the algorithm. We will now explain the Blake et al. methods and explore their significance. These methods work best when the polynomial $f(x)$ which defines the field has the special property that it divides some polynomial of the form

$$x^{2^k} + f_1(x) \ , \tag{5.1}$$

where the degree of $f_1(x)$ is very small, and where the primitive element $g = g(x) = x$. In general, it appears likely that the degree of $f_1(x)$ will be relatively high, which will make these new approaches of Blake et al. of little significance. In some cases, however, these methods produce startling improvements. This happens, for example, in the case of $n = 127$, when we take the defining polynomial to be $f(x) = x^{127} + x + 1$, since here

$$x f(x) = x^{2^7} + x^2 + x \ ,$$

and $f_1(x)$ has degree 2.

The first of the observations made by Blake and his collaborators is that if $f_1(x)$ is of low degree, the polynomials x^{2^r}, $1 \le r \le n-1$, will often have low degree when reduced modulo $f(x)$. When this degree is low enough to make that polynomial a product of polynomials from S, we obtain a linear equation of the desired kind, since $\log_x x^{2^r} = 2^r$. As an example, for $n = 127$ and $f(x) = x^{127} + x + 1$, we find that for $7 \le i \le 126$,

$$x^{2^i} = (x^{2^7})^{2^{i-7}} = (x^2 + x)^{2^{i-7}} = x^{2^{i-6}} + x^{2^{i-7}} \ ,$$

and repeated application of this result shows that each x^{2^r}, $0 \le r \le 126$, can be expressed in the form

$$\sum_{i=0}^{6} \varepsilon_i x^{2^i} \ , \quad \varepsilon_i = 0, 1 \ ,$$

and so the logarithms of all such elements can be quickly computed, and are of the form 2^r for some r. Furthermore, since

$$1+x^{2^r} = (1+x)^{2^r} = x^{127.2^r} ,$$

one can also obtain the logarithms of all elements of the form

$$\varepsilon_{-1} + \sum_{i=0}^{6} \varepsilon_i x^{2^i} , \quad \varepsilon_i = 0, 1 .$$

In particular, these will include the logarithms of 31 nonzero polynomials of degrees ≤ 16. In general, for other values of n, $f_1(x)$ will not have such a favorable form, and we can expect fewer usable equations.

Another observation of Blake et al., which is even more fruitful, is based on the fact that if $u(x)$ is any irreducible polynomial over $GF(2)$ of degree d, and $v(x)$ is any polynomial over $GF(2)$, then the degrees of all irreducible factors of $u(v(x))$ are divisible by d. To prove this, note that if $w(x)$ is an irreducible factor of $u(v(x))$, and α is a root of $w(x) = 0$, then $v(\alpha)$ is a zero of $u(x)$, and thus is of degree d over $GF(2)$. Since $v(x)$ has its coefficients in $GF(2)$, this means that α must generate an extension field of $GF(2^d)$, which means that its degree must be divisible by d, as we wished to show.

To apply the above fact, Blake et al. take an irreducible $u(x)$ of low degree, $u(x) \in S$, and note that by (5.1),

$$u(x)^{2^k} = u(x^{2^k}) = u(f_1(x)) .$$

If $u(f_1(x))$ factors into polynomials from S, one obtains another equation for the logarithms of the $v \in S$. The result proved in the preceding paragraph shows that all the factors of $u(f_1(x))$ will have degrees divisible by deg $u(x)$, and not exceeding (deg $u(x)$) (deg $f_1(x)$). Blake and his collaborators noted that in many cases all the irreducible factors have degrees actually equal to deg $u(x)$. We will now discuss the likelihood of this happening.

Suppose that

$$f(x) \mid x^{2^k} + f_1(x) . \tag{5.2}$$

We can assume without loss of generality that not all powers of x appearing in $f_1(x)$ are even, since if they were, say $f_1(x) = f_2(x^2) = f_2(x)^2$, we would have

$$f(x) \mid x^{2^k} + f_2(x)^2 = (x^{2^{k-1}} + f_2(x))^2 ,$$

and since $f(x)$ is irreducible, we would obtain

$$f(x) \quad \Big| \quad x^{2^{k-1}} + f_2(x) \quad ,$$

and we could replace $f_1(x)$ by $f_2(x)$ in (5.2). Therefore we will assume $f_1(x)$ does have terms of odd degree, and so $f_1'(x) \neq 0$.

The polynomial

$$F_d(x) = x^{2^d} + x \tag{5.3}$$

is the product of all the irreducible polynomials of all degrees dividing d. When we substitute $f_1(x)$ for x in $F_d(x)$, we obtain

$$F_d(f_1(x)) = f_1(x)^{2^d} + f_1(x) = f_1(x^{2^d}) + f_1(x) \ . \tag{5.4}$$

But

$$f_1(x^{2^d}) + f_1(x) \equiv f_1(x) + f_1(x) = 0 \quad (\text{mod } F_d(x)) \ ,$$

and so each irreducible polynomial whose degree divides d has to divide some $u(f_1(x))$ for another irreducible $u(x)$ of degree dividing d. Since

$$\frac{d}{dx} \ F_d(f_1(x)) = f_1'(x)$$

by (5.4), only a small number of irreducibles can divide $F_d(f_1(x))$ to second or higher powers. Hence we conclude that at most only about one in deg $f_1(x)$ of the irreducible polynomials $u(x)$ of degree d can have the property that $u(f_1(x))$ factors into irreducible polynomials of degree d. Thus if we have only one pair $(k, \ f_1(x))$ for which (5.2) holds, then we can expect at most about $|S|/(\deg \ f_1(x))$ systematic equations from this method. We also obtain useful equations from all $u(x)$ for which deg $u(f_1(x)) \leq m$, but there are relatively few such polynomials $u(x)$. If deg $f_1(x) = 2$ (as it is for $n = 127$, $f(x) = x^{127} + x + 1$), it is easy to see that almost exactly one half of the irreducible polynomials $u(x)$ of a given degree d will have the property that $u(f_1(x))$ factors into irreducible polynomials of degree d. If deg $f_1(x) > 2$, the situation is more complicated, in that the $u(f_1(x))$ can factor into products of irreducible polynomials of several degrees, and so the number of useful equations obtained this way is typically considerably smaller than

$|S|/(\deg\ f_1(x))$.

One factor which is hard to predict is how small can one take the degree of $f_1(x)$ so that (5.2) holds for some k and some primitive polynomial $f(x)$ of degree n. The situation for $n = 127$, where we can take $f_1(x) = x^2 + x$, is extremely favorable. For some n, it is possible to take $\deg\ f_1(x) = 1$. Condition (5.2) with $f_1(x) = x$ is not useful, since it holds precisely for the irreducible polynomials of degrees dividing k, and the resulting discrete logarithm equations simply say that

$$2^k \log_x v \equiv \log_x v \pmod{2^d - 1}$$

for $d | k$, $d = \deg\ v(x)$, which is trivial. Condition (5.2) with $f_1(x) = x+1$ is somewhat more interesting. If it holds, then

$$f(x) \mid x^{2^{2k}} + x ,$$

and thus $\deg\ f(x) \mid 2k$. On the other hand, because of (5.2), $\deg\ f(x) \nmid k$. Thus this condition can hold only for even n, which, as we will argue later, ought to be avoided in cryptographic applications. For these even n, however, it gives relations of the form

$$2^{n/2} \log_x v \equiv \log_x v^* \pmod{2^n - 1} ,$$

for all irreducible $v(x)$, where $v^*(x) = v(x+1)$, and then gives about $|S|/2$ useful equations.

In many cases it is impossible to find $f(x)$ of a given degree such that (5.2) holds for some $f_1(x)$ of low degree. When such $f(x)$ can be found, it sometimes happens that (5.2) holds for several pairs $(k,\ f_1(x))$. For example, when $n = 127$, $f(x) = x^{127} + x + 1$, condition (5.2) holds for $k = 7$, $f_1(x) = x^2 + x$ and also for $k = 14$, $f_1(x) = x^4 + x$.

The significance of these systematic equations is not completely clear. Our arguments indicate that unless (5.2) is satisfied with $f_1(x)$ of low degree, few systematic equations will be obtained. No method is currently known for finding primitive $f(x)$ of a given degree n for which (5.2) is satisfied with some $f_1(x)$ of low degree. It is not even known whether there exist such $f(x)$ for a given n. Even in the very favorable situation that arises for $n = 127$, $f(x) = x^{127} + x + 1$, Blake et al. [8] found only 142 linearly independent systematic equations involving the 226 logarithms of the irreducible polynomials of degrees ≤ 10. (They reported a very large number of linear dependencies among the systematic equations they obtained.) Thus it seems that while systematic equations are a very important idea that has already led to the Coppersmith breakthrough and might lead to further developments, at this time they cannot be relied upon to produce much more than $|S|/2$ equations, and in practice probably many fewer can be expected.

5.2 Change of primitive element and field representation

The Coppersmith algorithm requires that the polynomial $f(x)$ that generates the field $GF(2^n)$ be of the form (4.18) with $f_1(x)$ of low degree. Section 4.1 showed that if the $f(x)$ satisfies (5.2) with $f_1(x)$ if low degree, and x is a primitive element of the field, one can obtain many systematic equations. On the other hand, it is often desirable that $f(x)$ satisfy other conditions. For example, if $f(x)$ is an irreducible trinomial,

$$f(x) = x^n + x^k + 1 \ , \tag{5.5}$$

where we may take $k \leq n/2$, since $x^n + x^{n-k} + 1$ is irreducible if and only if $f(x)$ is, then reduction of polynomials modulo $f(x)$ is very easy to implement; if

$$h(x) = \sum_{i=0}^{2n-2} a_i x^i$$

(as might occur if $h(x)$ is the product of two polynomials reduced modulo $f(x)$), then

$$h(x) \equiv \sum_{i=0}^{n-1} a_i x^i + \sum_{i=0}^{n-2} a_{i+n} x^i + \sum_{i=k}^{k+n-2} a_{i+n-k} x^i \pmod{f(x)} \ , \tag{5.6}$$

a reduction that can be accomplished using two shifts and two exclusive or's of the coefficient strings, and another iteration of this procedure applied to the polynomial on the right side of (4.6) yields the fully reduced form of $h(x)$. It is often also desirable that $f(x)$ be primitive, since then x can be used as a primitive element of the field. (Extensive tables of primitive trinomials are available, see [28,71,72].) In some cases, of which $n = 127$ and $f(x) = x^{127} + x + 1$ is the example par excellence, it is possible to satisfy all these desirable conditions. In general, though, some kind of compromise might be necessary, and the choice to be made might depend both on n (and thus on what kinds of polynomials exist) and on the hardware and software that are being used. Our purpose here is to show that the security of a cryptosystem is essentially independent of the choices that are made; the cryptosystem designer and the cryptanalyst can choose whichever $f(x)$ and $g(x)$ suit them best.

To show that changing only the primitive element $g(x)$ does not affect the security of a system, suppose that we have a way to compute discrete logarithms to base $g(x)$ efficiently. If another primitive element $g_1(x)$ and a nonzero polynomial $h(x)$ are given, and it is desired to compute the logarithm of $h(x)$ to base $g_1(x)$, we compute the logarithms of $g_1(x)$ and $h(x)$ to base $g(x)$, say

$$g_1(x) \equiv g(x)^a \pmod{f(x)} \ ,$$

$$h(x) \equiv g(x)^b \pmod{f(x)} \ ,$$

and obtain immediately

$$h(x) \equiv g_1(x)^{a^* b} \pmod{f(x)} \ ,$$

where a^* is the integer with $1 \le a^* \le 2^n - 1$ for which

$$aa^* \equiv 1 \pmod{2^n - 1} \ .$$

(Since $g(x)$ and $g_1(x)$ are primitive, $(a_1 2^n - 1) = 1$, and so a^* exists.)

Changing the representation of the field, so that it is given as polynomials modulo $f_1(x)$, as opposed to modulo $f(x)$, also does not affect the difficulty of computing discrete logarithms, as was first observed by Zierler [70]. The two fields are isomorphic, with the isomorphism being given by

$$x \pmod{f_1(x)} \to h(x) \pmod{f(x)} \ ,$$

where

$$f_1(h(x)) \equiv 0 \pmod{f(x)} \ .$$

Thus to construct the isomorphism we have to find a root $h(x)$ of $f_1(x)$ in the field of polynomials modulo $f(x)$. Such a root can be found in time polynomial in n [7,16,36,55,70], which establishes the isomorphism and enables one to transfer logarithm computations from one representation to another.

5.3 Faster generation and processing of test polynomials

As we described the basic index-calculus algorithm, the polynomials h^* are generated (in the first stage of the algorithm, say) by selecting a random integer s and reducing g^s modulo $f(x)$. Typically this involves on the order of $3n/2$ polynomial multiplications and reductions modulo $f(x)$. This work can be substantially reduced by choosing the h^* in succession, say $h_1^* = 1,\ h_2^*,\ h_3^*, \ldots$, with

$$h_{k+1}^* \equiv h_k^* v_s \quad (\mathrm{mod}\ f(x)) \ ,$$

where v_s is chosen at random from S. This requires only one polynomial multiplication (in which one factor, namely v_s, is of low degree) and one reduction. Since each h_k^* is of the form

$$h_k^* \equiv \prod_{v \in S} v^{a_v} \quad (\mathrm{mod}\ f(x)) \ ,$$

any time we find that both w_1 and w_2 have all their irreducible factors in S, we obtain another equation for the $\log_g v,\ \ v \in S$. Heuristic arguments and some empirical evidence [58] indicate that the sequence h_k^* ought to behave like a random walk in $GF(2^n) \backslash \{0\}$, which means that the modified algorithm ought to produce linear equations about as efficiently as the old one.

Once h^* is computed, the $(w_1,\ w_2)$ pair that satisfies (4.7) is produced by the extended Euclidean algorithm applied to the polynomials h^* and f, which are each of degree about n. It might be advantageous to decrease the cost of this relatively slow operation by generating several pairs $(w_1,\ w_2)$ that satisfy (4.7). This can be done by choosing $w_1 = \gamma_j$ and $w_2 = \alpha_j$ for several values of j such that (4.11) holds and the degrees of the w_i are not too far from $n/2$. As is shown in Appendix A,

$$p(r+s,\ m)\, p(r-s,\ m) \approx p(r,\ m)^2$$

for s small compared to r (for example, $p(105,\ 18)\, p(95,\ 18) = 1.07 \times 10^{-8}$, while $p(100,\ 18)^2 = 1.09 \times 10^{-8}$) so that if the neighboring pairs $(\gamma_j,\ \alpha_j)$ that satisfy (4.11) are independent with regard to factorization into small degree irreducible polynomials, as seems reasonable, we can cheaply obtain additional pairs $(w_1,\ w_2)$ satisfying (4.7) which will be just as good in producing additional equations.

The two modifications suggested above can also be applied to the second stage of the basic index-calculus algorithm, where they will lead to a similar improvements in running time. They can also be used in the first step of the second stage of the Coppersmith algorithm.

Blake et al. [8] used the Berlekamp algorithm [7] to factor the polynomials w_i. However, what is really needed initially is only to check whether all the irreducible factors of the w_i are of degrees $\leq m$. The complete factorization of the w_i is needed only when the w_i are both composed of low degree factors, and this happens so infrequently that the time that is needed in those cases to factor the w_i is an insignificant fraction of the total running time. Now to rapidly check whether a polynomial $w(x)$ has all its irreducible factors of degrees $\leq m$, we can proceed as follows. Since the greatest common divisor, $(w'(x), w(x))$, of $w(x)$ and its derivative equals

$$(w'(x),\ w(x)) = \prod_i y_i(x)^{2[a_i/2]} \ , \tag{5.7}$$

where

$$w(x) = \prod_i y_i(x)^{a_i} \ ,$$

and the $y_i(x)$ are distinct irreducible polynomials, we can compute

$$w^{(0)}(x) = \prod_i y_i(x)$$

in a few greatest common divisor and square root operations. Then, for $i = 1, 2, \ldots, m$ we compute

$$w^{(i)}(x) = \frac{w^{(i-1)}(x)}{(w^{(i-1)}(x),\ x^{2^i} + x)} \ . \tag{5.8}$$

Since $x^{2^k} + x$ is the product of all the irreducible polynomials of degrees dividing k, $w^{(m)}(x) = 1$ if and only if all the irreducible factors of $w(x)$ are of degrees $\leq m$.

The above procedure ought to be quite fast, since the greatest common divisor of two polynomials of degrees $\leq n$ can be computed using at most n shifts and exclusive or's of their coefficient sequences and since the degrees of the $w^{(i)}$ are likely to decrease rapidly. The above procedure can be simplified some more by noting that it suffices to define $w^{(i_0)}(x) = w^{(0)}(x)$ for $i_0 = [(m-1)/2]$ and apply (5.8) for $i = i_0 + 1, \ldots, m$, since any irreducible polynomial of degree d, $d \leq m$, divides at least one of the $x^{2^i} + x$, $i_0 + 1 \leq i \leq m$. Furthermore, the $x^{2^i} + x$ do not have to be computed at each stage separately, but instead, if we save

$$u_i(x) \equiv x^{2^i} + x \pmod{w^{(i-1)}(x)} \ ,$$

with $u_i(x)$ reduced modulo $w^{(i-1)}(x)$, then

$$u_i(x) \equiv x^{2^i} + x \pmod{w^{(i)}(x)} ,$$

and so

$$u_{i+1}(x) \equiv u_i(x^2) + x^2 + x \pmod{w^i(x)} ,$$

which is a much simpler operation.

Another fast way to test whether a polynomial $w(x)$ has all its irreducible factors of degrees $\leq m$ was suggested by Coppersmith [19]. It consists of computing

$$w'(x) \prod_{i = \lceil m/2 \rceil}^{m} (x^{2^i} + x) \pmod{w(x)} ,$$

and checking whether the resulting polynomial is zero or not. This method avoids the need for many greatest common division computations, and so may be preferable in some implementations. It is not completely foolproof, since polynomials in which all irreducible factors of degrees $> m$ appear to even powers will pass the test. However, such false signals will occur very infrequently, and will not cause any confusion, since polynomials $w(x)$ that pass the Coppersmith test have to be factored in any case.

5.4 Large irreducible factors

This section discusses a variation on both the basic index-calculus algorithm and the Coppersmith variation that was inspired by the "large prime" variation on the continued fraction integer factoring method (cf. [53]). In practice, as will be discussed in greater length later, the w_i would probably be factored by removing from them all irreducible factors of degree $\leq m$, and discarding that pair (w_1, w_2) if either one of the quotients is not 1. If one of the quotients, call it $u(x)$, is not 1, but has degree $\leq 2m$, then it has to be irreducible. The new variation would use such pairs, provided the degree of $u(x)$ is not too high ($\leq m+6$, say). The pair (w_1, w_2) that produced $u(x)$ would be stored, indexed by $u(x)$. Then, prior to the linear equation solving phase, a preprocessing phase would take place, in which for each irreducible $u(x)$, deg $u(x) > m$, the pairs (w_1, w_2) that are associated to it would be used to obtain additional linear equations involving logarithms of the $v \in S$. For example, in the basic algorithm, if there are k pairs associated to $u(x)$, say

$$h_i^* \equiv u^{a_i} \prod_{v \in S} v^{b_v(i)} \pmod{f} \ , \quad 1 \le i \le k \ ,$$

where each $a_i = \pm 1$, then we can obtain $k-1$ equations for the logarithms of the $v \in S$ by considering the polynomials

$$h_i^* (h_1^*)^{-a_i/a_1} \equiv \prod_{v \in S} v^{b_v(i) - b_v(1) a_i / a_1} \pmod{f} \ , \quad 2 \le i \le k \ .$$

A similar method works with the Coppersmith variation.

We now consider the question of how many equations we are likely to obtain by this method. Suppose that we generate N different pairs (w_1, w_2), where each of the w_i is of degree approximately M (which would be $\sim n/2$ for the basic algorithm and on the order of $n^{2/3}$ in the Coppersmith variation). We then expect to obtain about

$$Np(M,\ m)^2$$

pairs (w_1, w_2), where each of the w_i factors into irreducibles from S. Consider now some $k > m$. The probability that a random polynomial of degree $\sim M$ has exactly one irreducible factor of degree k and all others of degrees $\le m$ is about

$$p(M-k,\ m) I(k) 2^{-k} \ ,$$

where $I(k)$ is the number of irreducible polynomials of degree k. Therefore we expect that the probability that exactly one of w_1 and w_2 has one irreducible factor of degree k and all other factors of both w_1 and w_2 are of degrees $\le m$ is about

$$2p(M-k,\ m) p(M,\ m) I(k) 2^{-k} \ .$$

(The probability that both w_1 and w_2 have one irreducible factor of degree k and all others of degree $\le m$ is negligible.) Hence among our N pairs (w_1, w_2) we expect about

$$N_k \sim 2N\ p(M,\ m) p([n/2]-k,\ m) I(k) 2^{-k} \tag{5.9}$$

pairs that would be preserved. The number of equations that we expect to obtain from these N_k pairs is $N_k - M_k$, where M_k is the number of irreducible polynomials of degree k that appear in the stored list.

To estimate M_k, we make the assumption that the irreducible polynomials $u(x)$ of degree k that appear in the factorization of the w_i behave as if they were drawn at random from the $I(k)$ such polynomials. When N_k balls are thrown at random into $I(k)$ buckets, the expected number of buckets that end up empty is $I(k)$ times the probability that any single bucket ends up empty. Since the probability that a particular bucket ends up with no balls is

$$\frac{(I(k)-1)^{N_k}}{I(k)^{N_k}} \ ,$$

the expected number of buckets that we expect to be occupied is

$$I(k) - I(k)(I(k)-1)^{N_k} \ I(k)^{-N_k} \ .$$

Therefore we expect to obtain approximately

$$N_k \ + \ I(k)((1-I(k)^{-1})^{N_k} - 1) \tag{5.10}$$

additional equations from polynomials of degree k. Since N_k will be comparable to $I(k)$ in magnitude in applications to the index-calculus algorithm, we can approximate (5.10) by

$$N_k \ + \ I(k) \ (\exp(-N_k/I(k)) - 1) \ . \tag{5.11}$$

Since (see Appendix A) $I_k \sim 2^k k^{-1}$ and

$$p(M-k, \ m) \sim p(M, \ m) \ (Mm^{-1} \ \log_e \ M/m)^{k/m} \ ,$$

(5.9) gives us

$$N_k \sim 2 \ Nk^{-1} p(M, \ m)^2 \ (Mm^{-1} \ \log_e \ M/m)^{k/m} \ . \tag{5.12}$$

Since $|S| \sim 2^{m+1} m^{-1}$, we are interested in N for which $Np(M, \ m)^2$ is on the order of $2^m m^{-1}$. For such N, though, (5.11) and (5.12) show that the number of additional equations is negligible for $k - m \to \infty$. For $k \sim m$, on the other hand, (5.12) shows that

$$N_k \sim 2 \ M \ m^{-2} N \ p(M, \ m)^2 \ (\log_e \ M/m) \ ,$$

which is

$$\sim c' Np(M, \ m)^2$$

for $m \sim c(M \log_e M)^{1/2}$, which is the case for both the basic algorithm and the Coppersmith variant. Hence we also have

$$N_k \sim c'' I(m) \ ,$$

and (5.11) then shows that we can expect

$$[c'' - 2^{k-m} (1 - \exp(-c'' 2^{m-k}))] I(m)$$

additional equations, where the implied constants are absolute. Hence when we sum over k, we find that the total number of additional equations we can expect the large irreducible factor variation to generate is proportional to the number that have to be obtained.

The large irreducible factor variation can be quite important for moderate values of n, especially when m is relatively low, as it might have to be to make the solution of the system of linear equations feasible. For example, for $M \sim 65$, $m = 18$, without the large irreducible factor variation we might expect to test about $N \approx 1.04 \times 10^8$ pairs (w_1, w_2), whereas with this variation we expect to need only about 6.7×10^7. For $M \sim 65$ and $m = 12$, the difference is even more dramatic, since without the variation we expect to need $N \approx 1.3 \times 10^{10}$, while with it we need only $N \approx 3.5 \times 10^9$. For $M \sim 100$ and $m = 20$ the figures are $N \approx 4.9 \times 10^{11}$ and $N \approx 2.3 \times 10^{11}$, respectively, while for $M \sim 100$ and $m = 18$ they are $N \approx 2.7 \times 10^{12}$ and $N \approx 1.1 \times 10^{12}$. Thus for values that are of cryptographic significance, the large irreducible variation can shorten the running time of the equation generating phase by a factor of between 2 and 3. Furthermore, it can speed up the second stage of the index-calculus algorithm by an even greater factor, since in addition to the logarithms of the $v \in S$, the cryptanalyst will possess the logarithms of many polynomials of degrees $m+1,\ m+2, \ldots$.

5.5 Early abort strategy

Like the large irreducible factor variation discussed in the preceding section, the early abort strategy is also inspired by a similar technique used in factoring integers. Most of the pairs (w_1, w_2) that are generated turn out to be ultimately useless, whether the large irreducible factor variation is used or not. It would obviously be of great advantage to be able to select those pairs (w_1, w_2) in which both of the w_i are likely to factor into irreducible polynomials from S. The idea behind the early abort strategy is that a polynomial is unlikely to have all its factors in S unless it has many factors of small degree. Asymptotically this variation is unimportant, since factorization of binary polynomials can be accomplished in time polynomial in their degree. For small values of n, though, this variation can be important, as will be shown below.

Let $p_k(r, m)$ denote the probability that a polynomial of degree r has all its irreducible factors of degrees strictly larger than k but at most m. It is easy to obtain recurrences for $p_k(r, m)$ similar to those for $p(r, m)$ derived in Appendix A, which enables one to compute the $p_k(r, m)$ numerically. (It is also possible to obtain asymptotic expansions for the $p_k(r, m)$, but since we know a priori that the early abort strategy is unimportant asymptotically, we will not do it here.) For a polynomial $w(x)$, let $w^*(x)$ denote the product of all the irreducible factors of $w(x)$ of degrees $\leq k$ (with their full multiplicity). Let $Q(r, R, m, k)$ denote the probability that a polynomial $w(x)$ of degree r has all its irreducible factors of degrees $\leq m$, that deg $w^*(x) \geq R$. Then we easily obtain

$$Q(r, R, m, k) = \sum_{j \geq R} p(j, k) p_k(r-j, m) .$$

Let $Q^*(r, R, k)$ denote the probability that a random polynomial $w(x)$ of degree r has the property that deg $w^*(x) \geq R$. Then we similarly obtain

$$Q^*(r, R, k) = \sum_{j \geq R} p(j, k) p_k(r-j, r-j) .$$

The early abort strategy with parameters (k, R) is to discard the pair (w_1, w_2) if either $w_1^*(x)$ or $w_2^*(x)$ has degree $< R$. Let A represent the time needed to check whether both $w_1(x)$ and $w_2(x)$ have all their irreducible factors are of degrees $\leq m$, and let B represent the time involved in testing whether the degrees of $w_1^*(x)$ and $w_2^*(x)$ are both $\geq R$. Then to obtain one factorization that gives a linear equation for the logarithms of the $v \in S$, the standard index-calculus algorithm has to test about $p([n/2], m)^{-2}$ pairs (w_1, w_2) at a cost of approximately

$$A\ p([n/2], m)^{-2} \tag{5.13}$$

units of time. The early abort strategy has to consider about $Q([n/2], R, m, k)^{-2}$ pairs (w_1, w_2), but of these only about $Q^*([n/2], R, k)^2\ Q([n/2], R, m, k)^{-2}$ pairs have to be subjected to the expensive test of checking if all their irreducible factors have degrees $\leq m$. Hence the work involved in obtaining an additional linear equation under the early abort strategy is about

$$\{B + A\ Q^*([n/2],\ R,\ k)^2\} \quad Q([n/2],\ R,\ m,\ k)^{-2}\ . \tag{5.14}$$

In Table 1 we present some values of the ratio of the quantity in (5.14) to that in (5.13):

Table 1. Evaluation of the early abort strategy.

n	m	k	R	ratio of (5.14) to (5.13)
128	16	4	5	$2.47\ B/A\ +\ 0.412$
128	16	5	5	$1.73\ B/A\ +\ 0.452$
200	20	4	5	$2.67\ B/A\ +\ 0.445$
200	20	5	6	$2.32\ B/A\ +\ 0.396$

We see from this that if $B/A \lesssim 1/10$, then one can reduce the work required to obtain an additional equation by 30-40%, which might speed up the algorithm by a factor of approximately 1.5.

The success of the early abort strategy is crucially dependent on the ability to quickly find the divisors w_i^* of the w_i that are composed only of irreducible factors of degrees $\leq k$. If we use the procedure suggested in Section 5.3, this can be accomplished quite easily. Given a polynomial $w(x)$ to be tested, we compute its square-free part $w^{(0)}(x)$ and go through the first k steps of the procedure described by (5.8). If $k = 4$, this can be simplified further. Here we only need to know

$$(w^0(x),\ x^8 + x) \quad \text{and} \quad (w^0(x),\ x^{16} + x)\ ,$$

and these can be computed by reducing $w^{(0)}(x)$ modulo $x^8 + x$ and modulo $x^{16} + x$, respectively, and looking up the greatest common divisors in precomputed tables. We could then decide not to reject $w(x)$ if the difference of the degree of $w^{(0)}(x)$ and the sum of the degrees of the two divisors is small enough. It might also be advantageous to avoid computing $w^{(0)}(x)$ on the first pass, compute

$$(w(x),\ x^8 + x)\ , \quad (w(x),\ x^{16} + x)\ ,$$

and accept or reject $w(x)$ depending on how small the difference between the degree of $w(x)$ and the sum of the degrees of those factors is.

One can obtain some further slight gains by using additional conditions further along in the computation of the $w^{(i)}(x)$ defined by (5.8). It seems safe to say, though, that the early abort strategy is unlikely to speed up the linear equation collection phase of the index-calculus algorithm by more than a factor of 2 or so.

5.6 Faster generation of equations in Coppersmith's method

It is possible to significantly speed up the first stage of Coppersmith's variant of the index-calculus algorithm by applying some of the ideas that occur in the second stage of that version. Asymptotically, the improvements are not important, but in practice they are likely to be much more important than all the other variations we have discussed so far, and could speed up the equation-collecting phase of the algorithm by factors of 10 to 20 for $n = 127$, by up to 300 for $n = 521$, and by over 1000 for $n = 1279$.

The idea behind the new variation is that instead of selecting $u_1(x)$ and $u_2(x)$ to be any pair of relatively prime polynomials of degrees $\leq B$ each, we select them to increase the chances of $w_1(x)$ and $w_2(x)$ splitting into low degree irreducible factors. To do this, we select a pair $v_1(x)$ and $v_2(x)$ of polynomials of degrees $\leq B-1$ (but close to B) such that each is composed of irreducible factors of degrees $\leq m$. We then select $u_1(x)$ and $u_2(x)$ of degrees $\leq B$ so that $v_1(x) \mid w_1(x)$ and $v_2(x) \mid w_2(x)$. The divisibility condition gives us $\deg\ v_1(x) + \deg\ v_2(x) \leq 2B-2$ homogeneous linear equations for the $2B$ coefficients of $u_1(x)$ and $u_2(x)$, and so we obtain at least 3 nonzero solutions. Moreover, these solutions can be found very fast, by using gaussian elimination on the $GF(2)$ matrix of size $\leq 2B-2$ by $2B$.

When $u_1(x)$ and $u_2(x)$ are selected by the above procedure, the probability of $w_1(x)$ splitting into irreducible factors of degrees $\leq m$ ought to be close to $p(h,\ m)$, and the probability of $w_2(x)$ splitting in this way ought to be close to

$$p(h2^k - n + B(2^k-1) + \deg\ f_1(x),\ m)\ .$$

Since $B = O\ (n^{1/3}\ (\log_e\ n)^{2/3})$, the form of the asymptotic estimate for the probability of both $w_1(x)$ and $w_2(x)$ splitting is not affected by this improvement. In practice, however, the improvements can be vital.

Some care has to be used in the application of the idea proposed above. The first stage of the index-calculus algorithm requires the generation of $|S| \sim m^{-1} 2^{m+1}$ linearly independent equations. The equations generated by the basic version of the algorithm and by the Coppersmith variation are expected to be largely independent of the preceding ones (as long as there are $< |S|$ of them) on heuristic grounds, and this is confirmed by computational experience. That is not the case, however, with the variation proposed above, because in general many pairs $(v_1(x), v_2(x))$ will give rise to the same pair $(w_1(x), w_2(x))$. To circumvent this difficulty, we select B so that the standard Coppersmith algorithm without the variation proposed here would generate about $1.6\,|S|$ equations. (This involves increasing B by at most 1.) We then implement the present variation, with the new value of B. Essentially all of the $1.6\,|S|$ equations that would be generated by the standard Coppersmith algorithm can be generated by the new variation with appropriate choices of $v_1(x)$ and $v_2(x)$, and most can be generated in roughly the same number of ways. Hence we can again model this situation in terms of the ''balls into buckets'' problem described in Section 5.4; we have about $1.6\,|S|$ buckets corresponding to the equations we can possibly obtain, and we are throwing balls into them corresponding to the equations our variation actually produces. If we obtain about $1.6\,|S|$ equations all told, approximately $1.6(1-e^{-1})\,|S| > 1.01\,|S|$ of them will be distinct, and so it will be overwhelmingly likely that $|S|$ of them will be independent.

In our new variation we do not need to check whether $(u_1(x), u_2(x)) = 1$, and thus whether $(v_1(x), v_2(x)) = 1$. Therefore we can prepare beforehand a list of all polynomials of degrees $\leq B-1$ that are composed of irreducible factors of degrees $\leq m$, and this will generate a slight additional saving over the standard Coppersmith algorithm. (In order to take full advantage of the sparse matrix techniques of Section 5.7, it might be best to use only irreducible factors of degrees $\leq m-5$, say.) The effort needed to compute $u_1(x)$ and $u_2(x)$ (i.e., to solve a small linear system of equations), which is comparable to the work needed to test whether a polynomial has all its irreducible factors of degrees $\leq m$, can be amortized over more test polynomials by requiring that degrees of $v_1(x)$ and $v_2(x)$ be $\leq B-2$, since that will produce at least 15 nonzero solutions each time.

There are other ways to speed up the Coppersmith algorithm. One way would be to fix $2B+2-b$ of the coefficients of $u_1(x)$ and $u_2(x)$, where b is maximal subject to being able to store about 2^b small integers.

Then, for all irreducible polynomials $u(x)$ of degrees $\leq m$, one could quickly compute those choices of the remaining b coefficients for which $w_1(x)$ or $w_2(x)$ is divisible by $u(x)$. All that would need to be stored for each of the 2^b combinations would be the sum of the degrees of the divisors that were found. This variation, however, does not appear as promising as the one discussed above, and it would require some very novel architectures to implement it on a parallel processing machine of the type we will discuss later. Hence we do not explore this variation further.

A slight improvement on the basic idea of this section is to allow $v_1(x)$ and $v_2(x)$ to have different degrees, subject to the requirement that their sum be $\leq 2B-2$, so as to make the degrees of $w_1(x)/v_1(x)$ and $w_2(x)/v_2(x)$ more nearly equal.

Another modification to the Coppersmith algorithm was suggested by Mullin and Vanstone [48]. It consists of choosing $w_1(x)$ to be of the form

$$w_1(x) = u_1(x)\ x^{h-a} + u_2(x)$$

for $a = 1$ or 2, say, and selecting

$$w_2(x) \equiv w_1(x)^{2^k}\ x^b \pmod{f(x)}\ ,$$

where b is chosen so as to give small degree for $w_2(x)$ after reduction modulo $f(x)$. This might allow the use of slightly lower degree polynomials for $u_1(x)$ and $u_2(x)$ than would otherwise be required, since if $u_1(0) = 1$, the equations this method yields ought to be idependent of those the basic method produces. This modification can be combined with the others suggested here.

5.7 Sparse matrix techniques

So far we have concentrated on variations on the linear equation collection phase of the index-calculus algorithm. However, as we noted in Section 4, the difficulty of solving systems of linear equations seemed for a long time to be an important limiting factor on the algorithm and affected the asymptotic estimate of its running time. For example, in the basic algorithm, if the term 2^{2m} in (4.16) were replaced by 2^{rm} for any $r > 2$ ($r = 3$ corresponding to the use of gaussian elimination, for example), then the minimum of (4.16) would occur not at $m \sim c_1(n\ \log_e n)^{1/2}$, but at a smaller value, $m \sim c_1(r)\ (n\ \log_e n)^{1/2}$, and would be larger, with c_2 replaced by

$$c_2(r) = r(2(r-1))^{-1/2} (\log_e 2)^{1/2} .$$

In this section, though, we will show that the linear equations produced by the index-calculus algorithm can be solved in time essentially $|S|^2$, where $|S|$is roughly the number of equations.

The matrices of coefficients of the linear equations generated by the first stage of the index-calculus algorithm are special in that they are very sparse. The reason is that the coefficient vector of each equation is obtained by adding several vectors $(b_v(h))$, indexed by $v \in S$, coming from factorizations of polynomials

$$h = \prod_{v \in S} v^{b_v(h)} .$$

Since the polynomials h are always of degrees $< n$, there can be at most n nonzero $b_v(h)$, and so each equation has at most n nonzero entries. This is a very small number compared to the total number of equations, which is around $\exp(n^{1/3})$ or $\exp(n^{1/2})$. The literature on sparse matrix techniques is immense, as can be seen by looking at [4,6,11,27,61] and the references cited there. Many of the techniques discussed there turn out to be very useful for the index-calculus problem, even though we face a somewhat different problem from the standard one in that we have to do exact computations modulo $2^n - 1$ as opposed to floating point ones. In the worst case, the problem of solving sparse linear systems efficiently is probably very hard. For example, it is known that given a set of $0-1$ vectors $\mathbf{v}_1, \ldots, \mathbf{v}_r$, each of which contains exactly three 1's, to determine whether there is a subset of them of a given size that is dependent modulo 2 is *NP*-complete [35]. Thus we cannot hope to find the most efficient worst case algorithm. However, very efficient algorithms can be found.

There are several methods for solving the systems of linear equations that arise in the index-calculus algorithms that run in tone $O(N^{2+\varepsilon})$ for every $\varepsilon > 0$, where N is the number of equations. The first ones were developed by D. Coppersmith and the author from an idea of N. K. Karmarkar. This idea was to adapt some of the iterative algorithms that have been developed for solving real, symmetric, positive definite systems of equations [6,11,33,39]. For example, in the original version of the conjugate gradient method [33], in order to solve the system $Ax = y$, where A is a symmetric positive definite real matrix of size N by N, and y is a given real column vector of length N, one can proceed as follows. Let x_0 be an arbitrary vector of length N, and let $P_0 = r_0 = y - Ax_0$. The algorithm then involves $\leq N-1$ iterations of the following procedure: given x_i, r_i, and p_i, let

$$a_i = \frac{(r_i, r_i)}{(p_i, Ap_i)} \ , \tag{5.15a}$$

$$x_{i+1} = x_i + a_i p_i \ , \tag{5.15b}$$

$$r_{i+1} = r_i - a_i A p_i \ , \tag{5.15c}$$

$$b_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)} \ , \tag{5.15d}$$

$$p_{i+1} = r_{i+1} + b_i p_i \ . \tag{5.15e}$$

It can be shown [33] that if the computations are done to infinite precision, the algorithm will find $r_i = 0$ for some $i \leq N-1$, and $x = y_i$ will then solve the original system $Ax = y$.

There are several problems with trying to use the conjugate gradient method to solve the systems of linear equations that arise in the index-calculus algorithms. One is that the system is not symmetric, and one has to solve $Bx = y$ where B is not even a square matrix. This problem can be bypassed (as is well known, cf. [33]) by solving the system $Ax = z$, where $A = B^T B$ and $z = B^T y$. Since B will in general be of almost full rank, solutions to $Ax = z$ will usually give us solutions to $Bx = y$. The matrix A will not in general be sparse, but its entries do not have to be computed explicitly, since it is only necessary to compute the vectors Ap_i, and that can be done by multiplying p_i first by B and then B^T. The matrix B can be stored in the sparse form, with rows and columns being given by lists of positions and values of nonzero coefficients.

The main difficulty with the use of the conjugate gradient method is that the basic theory was based on minimizing a quadratic functional, and this does not apply in finite fields. However, as was suggested by Karmarkar, the most important property of the algorithm is that the direction vectors p_i are mutually conjugate (i.e., $(p_i, Ap_j) = 0$ for $i \neq j$), and this is a purely algebraic property. Therefore the algorithm will terminate after at most $n-1$ iterations and will find a solution unless at some stage a vector p_i is encountered such that $(p_i, Ap_i) = 0$. This cannot happen if A is a real positive-definite matrix and $p_i \neq 0$, but can occur over finite fields. If the computations are being done over a large finite field, the probability of this occurring if x_0 is choosen at random is quite low. If the field is small, say $GF(q)$ with small q, this probability is much more significant, and the way to avoid the problem is to choose x_0 to have entries in a larger field, say $GF(q^t)$ for some small $t \varepsilon Z^+$.

The adaptation of the conjugate gradient algorithm outlined above has been tested successfully by the author on some small systems. The advantages of the method include not only speed, since only about NQ operations in the field $GF(q^t)$ are required, where Q is the number of nonzero entries in B, and thus $O(\log N)$ or $O((\log N)^2)$ in our problems, but also very modest storage requirements, since aside from the matrix B it is necessary to store the vectors x_i, p_i, r_i for only two consecutive values of i at a time.

An algorithm due to Lanczos [39], somewhat different from the conjugate gradient algorithm, was similarly adapted by Coppersmith to solve the linear systems arising in the index-calculus algorithm. Coppersmith used that method to obtain another solution to the linear system that arose in the implementation of his attack on discrete logarithms in $GF(2^{127})$.

A more elegant method for dealing with the index-calculus linear systems was invented recently by Wiedemann [66]. Suppose first that we wish to solve for x in $Ax = y$, where A is a matrix of size N by N (not necessarily symmetric) over a field $GF(q)$. Let $v_0, v_1, \ldots, v_{2N}$ be vectors of length K, which might be 10 or 20, with v_j consisting of the first K coefficients of the vector $A^j y$. Since to compute the v_j we need only start with y and keep multiplying it by A, without storing all the vectors $A^j y$, we need only $O(KN)$ storage locations, each one capable of storing an element of $GF(q)$, and the number of $GF(q)$ operations to carry out this computation is $O(NQ)$. Now the matrix A satisfies a polynomial equation of degree $\leq N$:

$$\sum_{j=0}^{N} c_j A^j = 0 \ , \tag{5.16}$$

and therefore also for any $k \geq 0$,

$$\sum_{j=0}^{N} c_j A^{j+k} y = 0 \ . \tag{5.17}$$

Eq. (5.17) implies that any single component of the $v_0, \ldots, v_{2N}$ satisfies the linear recurrence with characteristic polynomial

$$\sum_{j=0}^{N} c_j z^j \ . \tag{5.18}$$

Given any sequence of length on the order of N, the Berlekamp-Massey algorithm [29,44,56] finds its minimal characteristic polynomial in $O(N^2)$ operations in the field $GF(q)$. Hence if we apply the Berlekamp-Massey algorithm to each of the K coordinates of the vectors $v_0 \ , \ldots , \ v_{2N}$, we will in $O(KN^2)$ steps obtain K polynomials whose least common multiple is likely to be the minimal polynomial of A. When we do find that minimal polynomial, and it is of the form (5.18) with $c_0 \neq 0$, then we can easily obtain the desired solution to $Ax = y$ from

$$\begin{aligned} y &= A^0 y = -c_0^{-1} \sum_{j=1}^{N} c_j A^j y \\ &= A \left[-c_0^{-1} \sum_{j=1}^{N} c_j A^{j-1} y \right] \ . \end{aligned} \tag{5.19}$$

If A is nonsingular, then $c_0 \neq 0$, as is easy to see. Conversely, if $c_0 \neq 0$, then A is nonsingular, since we can then write

$$A \sum_{j=1}^{N} c_j A^{j-1} = -c_0 I \ .$$

In general in index-calculus algorithms, we have to solve a system of the form $Ax = y$, where A is of size M by N, with $M > N$ (but $M - N$ small). One way to reduce to the earlier case of a nonsingular square matrix is to take a submatrix A' of A of size N by N, and apply the algorithm presented above to $(A')^T x = z$ for some random vector z. If A' turns out to be nonsingular, we can then go back and search for solutions to $A'x = y$, which is what we are interested in. If A' is singular, though, we will obtain a linear dependency among the rows of A'. This means we can discard one of the rows of A' that was involved in that dependency and replace it with another row of that part of A that has not been used yet. After a few steps, we ought to obtain a nonsingular A', and this will enable us to solve for x in $Ax = y$. Wiedemann [66] also has a deterministic algorithm, which may not be as practical, however.

We conclude this section by discussing some very simple methods for solving sparse systems of equations. Some of these methods can be used as a preliminary step before the application of Wiedemann's algorithm, say, since they serve to reduce the effective size of the matrix that has to be dealt with. In some cases these methods by themselves could be just about as efficient as the techniques described above. These methods are based on a simple observation that has often been made in the work on sparse matrices (cf. [6]), namely that if a matrix is noticeably sparser on one end than on the other, then it is better to start gaussian elimination from the sparse end. In our case, if we arrange the matrix of coefficients so that the columns correspond to polynomials $v \in S$ sorted by increasing degree, then the right side of the matrix will be very sparse. (If we use the fast version for generating the h^* that is presented in Section 5.3, it is necessary to choose the random $v_r \in S$ to have only low degrees for this to remain true.) To see just how sparse that matrix is, consider the Coppersmith algorithm in $GF(2^n)$, with k, m, and B chosen to satisfy (4.34a-c) with α satisfying (4.38), and β and γ satisfying $\beta = \gamma$ and (4.40). If we take $M \sim m^{-1} 2^m$, then the matrix of coefficients will have about $2M$ rows and $2M$ columns, with columns $M+1, \ldots, 2M$ (approximately) corresponding to the irreducible polynomials of degree m. We now consider those columns. Any row in the matrix comes from adding two vectors of discrete logarithm coefficients from factorization of two polynomials of degrees about $B \cdot 2^k$, both of which are divisible only by irreducible factors of degrees $\leq m$. The probability that a polynomial of degree $B \cdot 2^k$, which factors into irreducibles of degrees $\leq m$, also is divisible by a particular chosen irreducible polynomial of degree exactly m is approximately

$$\frac{2^{-m}\, p(B2^k - m, m)}{p(B2^k, m)} \ ,$$

which, by Lemma A.3 of Appendix A, is

$$\sim\ 2^{-m}\ m^{-1} B 2^k \log(B2^k/m)\ .$$

Therefore the probability that any particular entry in the last M columns of the matrix is nonzero is about

$$2^{-(m-1)}\ m^{-1} B 2^k \log(B2^k/m)\ . \tag{5.20}$$

(The factor 2 comes from the fact that we are adding two vectors.) For the choices of B, 2^k, and m that were specified, this becomes δM^{-1}, where

$$\delta = 2\alpha\gamma\beta^{-2}/3 = 2\beta^{-3/2}/3 = \log\ 2 = 0.6931....$$

(Exactly the same asymptotic result also applies to the basic index-calculus algorithm.) Therefore, by the ''balls into buckets'' model, we expect that with probability about

$$(1 - \delta M^{-1})^{2M} \approx \exp(-2\delta) = 1/4 ,$$

any column among the last M will contain only zeros. This means that about $M/4$ of the M irreducible polynomials of degree m will not appear in any of the factorizations and so the data base obtained from the first phase will be missing those values. More importantly, it means that it was not necessary to obtain all of the $2M$ equations, as $7M/4$ would have sufficed. (In fact fewer than $7M/4$, since with that few equations, the chances of obtaining a zero column would be even larger, and in addition we would also have some irreducible polynomials of degrees $m-1$, $m-2$, etc., which would not appear in the equations.) In addition, the probability of a particular column among the last M having just a single nonzero coefficient is about

$$2M \cdot \delta M^{-1} (1 - \delta M^{-1})^{2M-1} \approx 2\delta \exp(-2\delta) = (\log 2)/2 = 0.346...$$

Thus an additional 0.346M of the last M columns would have a single nonzero coefficient, so that we could remove those columns together with the rows in which those columns have nonzero coefficients, solve the remaining system, and then obtain the values of logarithms corresponding to the deleted columns by back substitution. (Occasionally a row might contain two nonzero coefficients which are the only such in their columns, which would prevent recovery of the values of the corresponding logarithms, but that is not a significant problem.) Furthermore, removal of those rows and columns would create more columns with only a single nonzero coefficient, so that the size of the matrix could be cut down by more than 0.35M. However, both simulations and heuristic arguments show that if we proceed to carry out standard gaussian elimination, proceeding from the sparse end, then very rapid fill-in occurs. Therefore one does have to be careful about algorithms that are used.

The above discussion of just how sparse the index-calculus algorithms matrices are was meant to motivate the following method. It will be helpful to explain it in terms of operating on the full matrix, although in practice the matrix would be stored in the sparse encoding, using lists of nonzero coefficients and their positions for rows and columns, just as in the case of the algorithms discussed above. The algorithm is as follows:

Step 1: *Delete all columns which have a single nonzero coefficient and the rows in which those columns have nonzero coefficients.*

Step 1 is repeated until there are no more columns with a single nonzero entry.

Step 2: *Select those αM columns which have the largest number of nonzero elements for some $\alpha > 0$. Call these columns "heavy," the others "light."*

A typical value of α might be 1/32. The entries in the "heavy" columns for every given row might be stored on a disk, with a pointer attached to the row list indicating the storage location. These pointers would have coefficients attached to them, which are set to 1 initially. The weight of a row is then defined as the number of nonzero coefficients in its "light" columns.

Step 3: *Eliminate variables corresponding to rows of weight 1 by subtracting appropriate multiples of those rows from other rows that have nonzero coefficients corresponding to those variables.*

During execution of Step 3, if u times row i is to be subtracted from row j, the pointers attached to row j are to have added to them the pointers of row i, with their coefficients multiplied by u. Step 3 is to be repeated until there are no more rows of weight 1. At the end of this process there are likely to be many more equations than unknowns. We can then perform the following operation.

Step 4: *If r rows are excess, drop the r rows with highest weight.*

We now iterate Step 1, and then Step 3. We then go on to the next procedure. Note that if a variable indexed by j, say, appears in rows of weights $2 \le w_1 \le w_2 \le \cdots \le w_k$, then eliminating that variable using a row of weight w_i will increase the number of nonzero entries in the matrix (after deletion of the row of weight w_i and the column corresponding to our variable) by

$$(w_i - 1)(k-1) - w_i - (k-1) = (w_i - 2)(k-1) - w_i \ . \tag{5.21}$$

Hence to minimize the amount of fill-in, we need to choose that variable and that w_i (which clearly equals w_1) for which (5.21) is minimized. Keeping track of this quantity is fairly easy if we use a priority queue data structure.

Step 5: *Eliminate that variable which causes the least amount of fill-in.*

The algorithm outlined above can be implemented to run very fast, and it reduces the problem of solving a roughly $2M$ by $2M$ system to that of solving an αM by αM system. What is perhaps most remarkable, if the original system is sufficiently sparse, only the first few steps of the algorithm are needed. For example, if the elements of the matrix are chosen independently at random, so that the probability of an entry in the last M columns being nonzero is δM^{-1}, in the next $M/2$ column is $2\delta M^{-1}$, etc., where $\delta \leq 0.85$ (compared to $\delta = 0.693...$ for the optimal case of Coppersmith's algorithm), and $\alpha = 1/32$, than Steps 1-4 of the algorithm are all that is needed, since by the time they are completed, there is nothing left of the "light" portion of the matrix. This result is confirmed by simulations (with systems of sizes up to 96,000) and by heuristic arguments.

The method presented above draws on ideas that are well known in the literature on sparse matrices (cf. [11]). Moreover, some of these ideas have already been used in the factoring integers and computing discrete logarithms. For example, J. Davenport in his computations related to Coppersmith's algorithm [19] used some heuristic methods to minimize fill-in. Such methods were also used during the execution of the Blake et al. [8] version of the basic index-calculus algorithm in $GF(2^{127})$. According to R. Mullin (private communication), the system of about 16,500 equations in about that many variables ($m = 17$ was used) was reduced by methods similar to those presented above to a system of size under 1000, which was then solved by ordinary gaussian elimination. Moreover, their procedure did not involve such tricks as always choosing the equation with fewest nonzero entries during elimination, which appear to result in dramatic improvements in performance. Therefore we expect these methods to be quite useful.

6. Practical and impractical implementations

Blake, Fuji-Hara, Mullin, and Vanstone [8] have successfully tested the basic index-calculus algorithm on fields up to $GF(2^{127})$. They estimated that with their VAX 11/780, a relatively slow minicomputer, it would have taken them many CPU months to carry out the first stage of the algorithm for $GF(2^{127})$ with $m = 17$. On the HEP, a powerful multiprocessor to which they obtained access, their implementation of the algorithm took about 8 hours for the first stage, of which about one hour was devoted to solving linear equations. (Their systematic equations method produced a substantial fraction of all the required equations.) Once the first stage is completed, the second stage is expected to take around 1 CPU hour per logarithm even on the VAX 11/780. On the IBM 3081K, Coppersmith estimated that the equation collecting phase for $GF(2^{127})$ would take around 9 hours with the basic algorithm. Using his own variation, Coppersmith was able to find all the necessary polynomials (for $m = 12$) in 11 minutes [19]. (The factorization of the polynomials to obtain the actual equations took 8 minutes, and solution of the equations took 20 minutes, but these tasks were performed with a general purpose symbolic manipulation program, and so could undoubtedly be speeded up very substantially.) Further speedups, perhaps by a factor of 30 to 50, could be obtained by combining the variation proposed in Section 5.6, which might gain a factor of 10 to 20, with those of sections 5.4 and 5.5, which together might gain a factor of 2 or 3. Using the Cray-1 might gain an additional factor of 10 or so, because it is perhaps 5 times faster than the IBM 3081K and because it could store and manipulate the test polynomials (of degrees $\lesssim 42$) in single words. Thus we can expect that with current supercomputers the equation collecting part of the first phase of the algorithm can be completed in around one second. Since the database produced by the algorithm is not very large (16,510 127-bit numbers for $m = 17$ in the basic algorithm and 747 numbers for $m = 12$ in the Coppersmith variation), this means that individual logarithms in $GF(2^{127})$ can now be computed even on personal computers. Therefore $GF(2^{127})$ ought to be regarded as completely unsuitable for cryptographic applications. Our intention here is to explore what other fields might be appropriate.

We first consider the basic algorithm. Although it has been made obsolete by the Coppersmith variation in applications to the fields $GF(2^n)$, it is worth analyzing in detail, since by comparing our estimates to actual running times we will obtain a better idea of how accurate the estimates are.

In Section 5 we presented briefly a number of variations on the basic index-calculus algorithm. These variations were not analyzed very carefully, since we were interested only in the order of magnitude of the improvements that can be obtained from such techniques. The general conclusion to be drawn from that section is that the time to generate the pairs (w_1, w_2), can probably be neglected. The work needed to obtain $|S|$equations is probably no more than and at least 1/5 of the work needed to test

$$|S|\ p([n/2],\ m)^{-2}$$

pairs (w_1, w_2) by the procedure outlined in Section 4.3 to see whether all the irreducible factors of each of the w_i are in *S*. To test each w_i takes about $m/2$ operations of the form (4.8), each of which involves a squaring modulo a polynomial of degree perhaps $n/3$ on average (since the degrees of the $w^{(i)}(x)$ will be decreasing, especially if we use the early abort strategy with additional test along the way to discard pairs (w_1, w_2) that are not factoring satisfactorily), a greatest common divisor operation or two polynomials of degrees around $n/3$, and a division, which will usually be trivial.

To evaluate the significance of the index-calculus algorithm for cryptographic schemes, we have to look at the effect of parallel processing and at speeds of modern circuits. We will assume that no exotic algorithms, such as fast integer multiplication using the Fast Fourier Transform [10] are to be used, since they are probably not practical for *n* on the order of several hundred. Since a cryptographic scheme ought to be several orders of magnitude too hard to break, we will only try to be accurate within a factor of 10 or so.

It appears that at present, custom VLSI chips could be built that would perform about 10^8 operations per second, where each operation would consist of a shift of a register of length 200 to 300 or else an exclusive or of two such registers. Semi-custom chips, which would be much easier to design and cheaper to produce, could operate at about 10^7 operations per second. Within the next decade or so, these speeds might increase by a factor of 10, so custom chips might do 10^9 operations per second, while semi-custom ones do 10^8. General purpose supercomputers like the Cray-1 can do about 10^8 operations per second when running in vector mode to take advantage of parallel processing, where each operation consists of a shift or exclusive or of 64-bit words. The structure of the index-calculus algorithm lends itself to parallel processing, but the fact that coefficients of polynomials would often take more than a single machine word to store would cause a substantial slowdown in operations, perhaps to a level of 10^7 operations per second. The next generation of supercomputers, such as the Cray-2, will be about 10 times faster, and might run at the equivalent of 10^8 operations per second.

The number of shifts and exclusive or's that are involved in squaring a polynomial of degree $\sim n/3$ modulo another polynomial of roughly that same degree and then in taking the greatest common divisor of two polynomials of degrees $\sim n/3$ can be roughly estimated by $3n$. Therefore each of the roughly $|S|\ p([n/2],\ m)^{-2}$ pairs $(w_1,\ w_2)$ that are generated can be expected to require about $3mn$ operations. (Various branchings and the like would make the actual algorithm slower, but this would be compensated somewhat by the factor of 3 or more that we might gain from using the large irreducible factor and the early abort variations, and the method of systematic equations. Note also that almost always it is only necessary to test w_1, since when it turns out not to factor in the desired way, there is no need to test w_2.) We therefore expect that about

$$n\ 2^{m+3} p([n/2],\ m)^{-2} \tag{6.1}$$

operations might be needed to generate the linear equations for the $\log_g v,\ \ v \in S$. Below we give approximations to the minimal values of (6.1) for various values of n as m varies (only values of $m \leq 40$ were considered):

Table 2. Operation count for the basic algorithm.

n	minimum of (6.1)	m
120	3.3×10^{11}	19
160	2.9×10^{13}	23
200	1.6×10^{15}	26
240	6.5×10^{16}	29
280	2.0×10^{18}	32
320	5.2×10^{19}	35
360	1.1×10^{21}	37
400	2.1×10^{22}	40
500	3.5×10^{25}	40

We will now temporarily neglect the effort needed to solve the linear equations that are generated, and discuss for which n and m one could hope to generate the required linear equations with various hardware configurations. We will assume that the equations are to be generated within one year, roughly 3×10^7 seconds. If we use a single supercomputer, we can hope to carry out between 3×10^{14} and 3×10^{15}

operations in that year. If we use a massively parallel machine with M special chips, we can expect to carry out between $3 \times 10^{14}\ M$ and $3 \times 10^{16}\ M$ operations in a year, depending on the technology that is used. Comparing these figures with those in the table in the preceding paragraph we see that even under our very optimistic assumptions, a general supercomputer could not assemble the required set of linear equations in under a year if $n \gtrsim 240$, say, whereas it probably could for $n \lesssim 180$. On the other hand, even a relatively modest special purpose processor using 10^4 semi-custom chips based on current technology could perform about 3×10^{18} operations per year, and so could probably cope with $n \gtrsim 260$, and perhaps with $n \gtrsim 280$, but probably not much beyond it. A very ambitious processor, using 10^6 custom designed chips operating at speeds that might become attainable in the next decade could do about 3×10^{22} operations per year, and could probably generate the needed equations for $n \lesssim 380$, but probably not for $n \gtrsim 420$.

The estimates made above are probably quite accurate, as is confirmed by comparing the numbers in Table 2 with the results of the $GF(2^{127})$ computations of [8]. Interpolating between the values in Table 2, we might expect that $GF(2^{127})$ might require about 10^{12} operations on a modern supercomputer, which is roughly what can be done in a day to a week. On the HEP, which is one of the modern multiprocessor supercomputers, the actual running time was about 7 hours, even though the method of systematic equations yielded about half of the equations practically for free.

The discussion in the preceding paragraph dealt only with the equation collection phase of the algorithm. The main reason for this is that the methods discussed in Section 5.7 make solving those equations rather negligible. However, in some cases this part of the algorithm might be nontrivial, since it would require doing arithmetic modulo $2^n - 1$. It is possible to envisage VLSI chips that multiply n-bit integers very fast, but such chips have impractically large areas. At the present time the best practical designs appear to be able to multiply two n-bit integers modulo another n-bit integer in about n clock periods (cf. [12]). Therefore we can expect that special purpose chips could perform between $n^{-1} 10^7$ and $n^{-1} 10^9$ multiplications modulo $2^n - 1$ per second, depending on the technology. In the case of a modern supercomputer, which could possibly perform about 10^8 multiplications on 32-bit words per second, we could expect about $10^8 / (10 (n/32)^2) \approx 10^{10} n^{-2}$ modular multiplications per second, and this will probably go up to $10^{11} n^{-2}$ in the next generation of supercomputers. (The factor 10 is there largely to compensate for the difficulty of working with multi-word numbers. We ignore the fact that many modern computers, such as the Cray-1, only allow 24-bit integer multiplication.)

In many situations, solving linear equations should be regarded as a limiting factor not so much due to its high operation count, but rather due to its requirements for a large memory and operation synchronization. A special purpose multiprocessor for the collection of equations is relatively simple to build. Each of the processors in it is quite simple, with essentially no storage, and these processors can operate independently of each other. Every once in a while one of these processors will find a factorization of the desired kind, which will then be sent to a central processor for storage. This also means that a multiprocessor of this kind would be fault-tolerant, since any factorization obtained by a small processor could be easily checked either by the central processor or by other processors without affecting the running time significantly. Therefore it would be very easy to build a multiprocessor to collect equations. On the other hand, a multiprocessor built for solving linear equations would require a very large memory, all the processors in it would have to operate synchronomsly under the control of the central unit, and it would have to operate essentially without errors. Such a multiprocessor would be much harder to build, and so we will often consider the use of a supercomputer for the equation solving phase together with a modest special purpose multiprocessor for the equation collecting phase.

In the case of the basic algorithm, the estimates derived from Table 2 for the running time of the algorithm do change somewhat if we consider using a modern supercomputer to solve the equations. For example, for $n = 400$, the value 2.1×10^{22} for the number of operations to find the needed equations requires the use of $m = 40$, which means that the number of unknowns (and equation) is around 5×10^{10}. Moreover, each equation might involve around 20 nonzero coefficients (which are usually equal to 1, though). Thus even with the use of the method described at the end of Section 5.7 to reduce the number of equations, of sorting on a disk, and sophisticated data structures, it seems that $m = 40$ would not be practical. However, use of $m = 35$ would reduce the size of the storage required by a factor of about 30, while increasing the number of operations to obtain the linear equations to only 3.5×10^{22}. Further reduction of m, to ≤ 30, would bring solution of the linear equations within practical reach without drastically increasing the effort needed for the equation collection phase.

The basic conclusion to be drawn from the preceding discussion is that using the basic algorithm, a supercomputer could probably be used to complete the first phase for $n \lesssim 200$, but almost certainly not for $n \gtrsim 300$. Using a relatively simple special purpose multiprocessor to assemble the equation and a supercomputer to solve them might be feasible for $n \lesssim 300$. Finally, even a very ambitious special purpose machine with 10^6 chips operating at 1 nanosecond per operation would not suffice for $n \gtrsim 500$.

The above discussion applied to the basic index-calculus algorithm. We next analyze the Coppersmith variation. In this case the performance of the algorithm can again be improved through use of the large irreducible factor variation and the early abort strategy, but again probably only by a factor of 3 to 5. Hence we will neglect these techniques. On the other hand, the method described in Section 5.6 leads to a speedup by two or three orders of magnitude, and so we will take it into account. As before, we first neglect the effort needed to solve the linear equations, and estimate only the work involved in finding those equations.

In the first stage of the Coppersmith algorithm, the time to generate the polynomials $w_1(x)$ and $w_2(x)$ can probably be neglected, especially since for each choice of $v_1(x)$ and $v_2(x)$ in the variation of Section 5.6 we will typically obtain several $(w_1(x), w_2(x))$ pairs. The main work consists of testing the pairs (w_1, w_2) to see whether all the irreducible factors of the w_i are in S. By a reasoning almost identical to that used in analyzing the basic algorithm (but with n replaced by $2h$), we see that this ought to take about $6mh$ exclusive or's and shifts. Hence the total number of such operations might be around

$$h 2^{m+4} p(h, m)^{-1} p(M, m)^{-1} , \tag{6.2}$$

with

$$M = \max \left(h 2^k - n + 2^k d_1 - d_2 + \deg f_1(x), (2^k - 1) d_2 \right) ,$$

where we select $\deg u_i(x) \approx \deg v_i(x) \approx d_i$, $i = 1, 2$. (There is occasionally some slight advantage in allowing different degree bounds for u_1 and u_2.) We also have to satisfy

$$p(h + d_1, m) p(M + d_2, m) 2^{d_1 + d_2 + 1} \gtrsim m^{-1} 2^{m+2}$$

in order to have enough possible equations.

In the table below we present approximate values for the minimal number of operations that these estimates suggest. In the preparation of this table, deg $f_1(x)$ was taken to be 10, since that is approximately what it is for such cryptographically important values of n as 521, 607, 881, and 1279. Also, only values of $m \le 40$ were considered.

Table 3. Operation count for Coppersmith's algorithm (equation collecting phase only).

n	approximate minimum of (6.2)	2^k	h	m	d_1	d_2
280	4.5×10^{11}	4	70	20	14	16
400	4.8×10^{13}	4	100	23	17	20
520	3.0×10^{15}	4	130	27	20	22
700	7.3×10^{17}	4	175	31	24	26
880	3.8×10^{18}	8	110	36	27	29
1060	6.0×10^{20}	8	133	38	29	31
1280	1.3×10^{22}	8	160	39	32	33

The above table dealt with the equation collection phase of Coppersmith's algorithm. As in the case of the basic algorithm, the equation solution phase would be limited more by the large memory size needed then by the number of operations. If we consider somewhat smaller values of m, we obtain Table 4.

There are two entries in the table for each n. It is possible to obtain finer estimates than in Table 4 by using what are in effect fractional values of m. What that would mean, in practice, is that S might consist of all the irreducible polynomials of degrees $\le m'$ and one third of those of degree $m'+1$, say. However, since Table 4 is meant to be used only as a rough guide, accurate only to within an order of magnitude, there is no point in doing this.

Table 4. Operation count for Coppersmith's algorithm (taking into account limitations of equation solving phase).

n	value of (6.2)	2^k	h	m	d_1	d_2
280	2.7×10^{12}	4	70	16	17	20
280	4.7×10^{11}	4	70	19	14	17
400	1.3×10^{14}	4	100	20	19	22
400	7.3×10^{13}	4	100	21	18	21
520	1.3×10^{16}	4	130	22	23	25
520	7.0×10^{15}	4	130	23	22	24
700	1.2×10^{19}	4	175	24	28	31
700	2.6×10^{18}	4	175	26	26	29
880	2.0×10^{21}	4	220	27	32	34
880	4.3×10^{20}	4	220	29	30	32
1060	2.4×10^{24}	8	133	30	38	40
1060	1.2×10^{23}	8	133	31	35	37
1280	4.3×10^{26}	8	160	31	43	44
1280	1.1×10^{24}	8	160	33	37	38
2000	1.7×10^{30}	8	250	36	48	50
2000	1.3×10^{29}	8	250	37	46	47

If we neglect the time needed to solve the system of linear equations, we see that a single supercomputer could probably compute the database for $n \lesssim 460$ in about a year, and the next generation might be able to do it for $n \lesssim 520$. On the other hand, $n \gtrsim 800$ would be safe from such attacks. If we assume that methods such as those of Section 5.7 are to be used to solve the linear equations, then Table 4 suggests that $n \gtrsim 700$ is safe even from the next generation of supercomputers, while $n \lesssim 500$ probably isn't.

A special purpose processor using 10^4 chips running at 100 nanoseconds per cycle might be able to assemble the equations for $n \lesssim 700$ in about a year, and these equations could probably be solved in about that much time on a supercomputer. For $n \approx 520$, though, a processor consisting of only about 100 chips of this kind might be able to find the equations in about a year (with $m = 22$), and they could then be solved in about a month on a supercomputer like the Cray-2. (Alternatively, with 10^3 chips in the equation collecting phase, a supercomputer might be needed for only a couple of days.) A very fanciful multiprocessor with 10^6 chips running at 1 nanosecond per cycle might be able to assemble the required equations for $n \lesssim 1280$ and solve them in between 1 and 10 years. Since even relatively small improvements to presently known algorithms could lower the operation count by a factor of 10 or 100, this means that even $n = 1279$ should not be considered safe, since it could then be broken using a less ambitious machine. (Note that a machine using 10^6 chips running at around 10 nanoseconds per cycle was proposed by Diffie and Hellman [24] for finding a DES key in about a day through exhaustive search. Such a machine was generally thought to be too ambitious for the then current technology, but it seems to be generally accepted that it could be built for some tens of millions of dollars by 1990.) On the other hand, $n \gtrsim 2200$ is about 10^6 times harder than $n \sim 1280$, and so can be considered safe, barring any new major breakthroughs in discrete logarithm algorithms.

7. Algorithms in $GF(p)$, p prime

The Silver-Pohlig-Hellman algorithm presented in Section 2 obviously applies directly to prime fields. The basic version of the index-calculus algorithm that has been presented so far can also be applied mutatis mutandis to the computation of discrete logarithms in fields $GF(p)$, p *a*prime. However, its simplest adaptation, even with the use of the early abort strategy [53], results in a running time for the first phase of about

$$L^{(5/2)^{1/2}} = L^{1.581...} \ , \tag{7.1}$$

where L stands for any quantity that is

$$L = \exp((1+o(1)) \ (\log_e p \ \log_e \log_e p)^{1/2}) \ \text{as} \ n \to \infty \ . \tag{7.2}$$

It was recently found, however, that there are several algorithms which run in time L [20]. The second phases of those algorithms can be used to find individual logarithms in time $L^{1/2}$ [20].

The discovery of the new algorithms for computing discrete logarithms in fields $GF(p)$ means that discrete logarithms in these fields are just about as hard to compute as it is to factor integers of size about p, provided that the field $GF(p)$ is changing. If the field were to stay fixed, then there would be an initial phase that would be about as hard to do as factoring a general integer around p, but then each individual logarithm would be relatively easy to compute.

Until recently, it was thought that the Schnorr-Lenstra algorithm [60] was the only factorization algorithm that ran in time L, with various other methods, such as the Pomerance quadratic sieve [53] requiring time $L^{1+\delta}$ for various $\delta > 0$. Those conclusions were based on the assumption that one had to use general matrix inversion algorithms to solve systems of linear equations. Now, with the methods described in Section 5.7 that take advantage of the sparseness of those systems, there are several algorithms, including the quadratic sieve and the Schroeppel linear sieve, and the new ones proposed in [20], which factor integers of size around p in time L.

It is quite possible that further progress in both discrete logarithm and factorization algorithms could be made in the near future. For example, if one can find, for a given p, integers a, b, and c such that they are all $O(p^{1/3+\epsilon})$ and such that

$$a^3 \equiv b^2 c \pmod{p}, \quad a^3 \neq b^2 c \;, \tag{7.3}$$

then one obtains a discrete logarithm algorithm and a factorization algorithm with running time

$$L^{(2/3)^{1/2}} = L^{0.8164...} \tag{7.4}$$

for the first phase [20]. Such a, b, and c are expected to exist for all p, and the problem is to construct an algorithm that finds them. In some cases they can be found. For example, if $p = a^3 - c$ for $c = O(p^{1/3})$, then (7.3) is satisfied with $b = 1$. (This version is the "cubic sieve" of Reyneri.) Any algorithm for constructing a, b, and c satisfying (7.3) would help about equally in factoring integers and computing discrete logarithms. In general, while there are algorithms for factorization that do not generalize to give discrete logarithm algorithms (the Schnorr-Lenstra algorithm [60], for example), the converse is not the case. Therefore it seems fairly safe to say that discrete logarithms are at least as hard as factoring and likely to remain so.

The idea behind the Coppersmith variant cannot be extended to the fields $GF(p)$ with p prime. That idea is based on the fact that squaring is a linear operation in $GF(2)$, so that if the difference of two polynomials over $GF(2)$ is of low degree, so is the difference of the squares of those polynomials. Nothing like this phenomenon seems to hold in the fields $GF(p)$, p prime.

8. Cryptographic implications

The preceding sections presented descriptions of the most important known algorithms for computing discrete logarithms in finite fields. The conclusions to be drawn from the discussion of these algorithms is that great care should be exercised in the choice of the fields $GF(q)$ to be used in any of the cryptosystems described in the Introduction. The Silver-Pohlig-Hellman algorithm presented in Section 2 has running time on the order of $\sqrt{p}$, where p is the largest prime factor of $q-1$. It is possible to decrease the $\sqrt{p}$ running time in cases where many discrete logarithms in the same field are to be computed, but only at the cost of a substantially longer preprocessing stage. Of the cryptosystems based on discrete logarithms, probably the most likely ones to be implemented are the authentication and key exchange ones (cf. [38,59,69]). To crack one of these systems, it is only necessary to compute one discrete logarithm, since that gives the codebreaker a valid key or password, with which he can then either impersonate a valid user or forge enciphered messages. Thus it can be expected that any discrete logarithm method would be used relatively infrequently in cryptanalysis, so that optimizing the form of the Silver-Pohlig-Hellman algorithm would yield both the preprocessing stage and the average running time on the order of $\sqrt{p}$, or at least within a factor of 100 or so of $\sqrt{p}$. The Silver-Pohlig-Hellman algorithm can be parallelized to a very large extent, the main limitation arising from the need to have a very large memory, on the order of $\sqrt{p}$ bits, which would be accessible from all the independent elements. This means that values of $p \lesssim 10^{25}$, say, ought to be avoided in cryptographic applications. On the other hand, for $p \gtrsim 10^{40}$, the Silver-Pohlig-Hellman algorithm appears impractical for the foreseeable future.

The limitation that $q-1$ have at least one large prime factor, which is imposed by the Silver-Pohlig-Hellman algorithm, has led to suggestions that fields $GF(2^n)$ be used for which 2^n-1 is a prime. Primes of the form $2^n - 1$ are known as Mersenne primes, and the known ones are listed in Table 5. One disadvantage of Mersenne primes is that there are relatively few of them. In particular, there are wide gaps between the consecutive Mersenne primes $2^{607} - 1$, $2^{1279} - 1$, and $2^{2203} - 1$. The index-calculus algorithm is not very sensitive to the factorization of $2^n - 1$, and so it seems safe to use values of n for which $2^n - 1$ is not prime, provided it has a large prime factor ($\gtrsim 10^{40}$, preferably, for reasons discussed above). Table 6 presents a selection of values of n between 127 and 521 for which the complete factorization of $2^n - 1$ is known and includes a very large prime factor. (This table is drawn from [14], except that the primality of the 105-digit factor of $2^{373} - 1$ was proved by the author using the Cohen-Lenstra [17] version of the Adleman-Pomerance-Rumely primality test [2].) Also included are the two values $n = 881$ and $n = 1063$, for which the cofactors have not been shown to be prime, although they almost definitely are, since they pass pseudoprime tests. Any one of these values of n will give a cryptosystem that is resistant to attacks by the Silver-Pohlig-Hellman algorithm.

It would be very desirable to have some additional entries in Table 6 to fill in the gap in Table 5 between $n = 1279$ and $n = 2203$. Unfortunately no prime values of n in that range are known for which $2^n - 1$ has been shown to contain a very large prime factor. It is possible to obtain composite values of n with this property (any multiple of 127 or 241 will do), but these are probably best avoided, since logarithms in these fields $GF(2^n)$ might be easy to compute. More generally, it might be advisable to avoid fields $GF(q)$ which have large subfields. Hellman and Reyneri [30] raised the possibility that the fields $GF(p^2)$ with p prime might be more secure than the fields $GF(p)$, since the index-calculus algorithm did not seem to extend to them. However, ElGamal [25] has shown how to modify the index-calculus algorithm to apply to most of the fields $GF(p^2)$. Furthermore, ElGamal's approach can be extended to all the fields $GF(p^2)$, and in fact to fields $GF(p^n)$ with n bounded. Thus fields of this kind appear not to offer increased security. In fact, these fields may be very weak because of the possibility of moving between the field and its subfields. As an example of the danger that exists, it can be shown that if $p + 1$ is divisible only by small primes, computing logarithms in $GF(p^2)$ is only about as hard as in $GF(p)$.

In the case of $GF(2^{127})$, the first stage of the index-calculus algorithm can now be carried out in a matter of hours on a minicomputer. Furthermore, once the database is generated, individual logarithms can

Table 5. Known Mersenne primes $2^p - 1$.

VALUES OF p FOR WHICH $2^p - 1$ IS PRIME
2
3
5
7
13
17
19
31
61
89
107
127
521
607
1,279
2,203
2,281
3,217
4,253
4,423
9,689
9,941
11,213
19,937
21,701
23,209
44,497
86,243
132,049

be computed rapidly even on today's personal computers. For that reason the field $GF(2^{127})$ should be regarded as very unsafe for cryptographic applications.

Once n moves up to 400 or so, the first stage of the index-calculus algorithm becomes infeasible to carry out in the fields $GF(2^n)$ by anyone not having access to computing resources comparable to those of a modern supercomputer. However, if somebody does use a supercomputer or a large number of smaller machines to carry out the first stage, and makes the database widely available, anyone with access to even a medium speed computer can rapidly compute individual logarithms.

When n reaches 700 or so, the first stage becomes infeasible even with a supercomputer. However, a relatively modest special purpose device consisting of about 10^4 semi-custom chips would enable a

Table 6. Factorization of Mersenne numbers $2^p - 1$ (p prime) which contain a very large prime factor. *Pn* denotes a prime of *n* decimal digits, *PRPn* a probable prime of *n* digits.

p	Factorization of $2^p - 1$
167	$2349023 \cdot P44$
197	$7487 \cdot P56$
227	$269863334377777017 \cdot P52$
241	$220000409 \cdot P66$
269	$13822297 \cdot P74$
281	$80929 \cdot P80$
307	$14608903 \cdot 85798519 \cdot 23487583308 \cdot$ $\cdot\ 78952752017 \cdot P57$
331	$16937389168607 \cdot 865118802936559 \cdot P72$
373	$25569151 \cdot P105$
409	$4480666067023 \cdot 76025626689833 \cdot P97$
881	$26431 \cdot PRP261$
1063	$1485761479 \cdot PRP311$

cryptanalyst to assemble the desired database even in these ranges. Such a special purpose computer might be assembled as part of a university project (cf. [62]). Furthermore, computations of individual logarithms could still be performed on any relatively fast computer. Special purpose machines of this kind, but either with more special chips or with faster chips could probably be used to assemble the databases for n up to perhaps 1200, but might have difficulty solving the system of linear equations.

The fields $GF(2^n)$ have been preferred for cryptographic applications because of ease of implementation. There are penalties for this gain, though. One is that the codebreaker's implementation is correspondingly easy to carry out. Another is that logarithms in the fields $GF(2^n)$ are much easier to compute than in the fields $GF(p)$ for p a prime, $p \sim 2^n$, especially now that the Coppersmith algorithm is available [18,19]. Still another disadvantage of the fields $GF(2^n)$ as compared with the prime fields of the same order is that there are very few of them. All of the fields $GF(2^n)$ with a fixed value of n are isomorphic, and so can be regarded as essentially the same field. On the other hand, there are many primes p with $2^{n-1} < p < 2^n$. This is important, since in the index-calculus algorithm (and to some extent also in the Silver-Pohlig-Hellman algorithm) the initial preprocessing stage has to be done only once, and once it's done, individual logarithms are computable relatively fast. If the field can be changed, say every month or every year, the cryptanalyst will have only that long on average to assemble his database. (This may not be a serious strengthening of security in the case of information that has to be kept secret for extended periods of time.) Therefore having only a few fields to choose from makes a cryptosystem less secure.

The algorithms presented here show that great care has to be exercised in the choice of the fields $GF(2^n)$ for cryptographic applications. First of all, n should be chosen so that $2^n - 1$ has a large prime factor, preferably larger than 10^{40}. Secondly, n should be quite large. Even to protect against attackers possessing small but fast computers of the kind that might be widely available within the next ten years, it seems best to choose $n \geq 800$. To protect against sophisticated attacks by opponents capable of building large special purpose machines, n should probably be at least 1500. In fact, to guard against improvements in known algorithms or new methods, it might be safest to use $n \geq 2000$ or even larger.

Given a bound on the size of the key, one can obtain much greater security by using the fields $GF(p)$ with p prime then the fields $GF(2^n)$. In this case p also has to be chosen so that $p - 1$ contains a large prime factor. If this precaution is observed, fields with $p \gtrsim 2^{750}$ provide a level of security that can only be matched by the fields $GF(2^n)$ with $n \gtrsim 2000$.

The requirement that p be changed frequently is not an onerous one. It is easy to construct large primes p for which $p - 1$ have a large prime factor (cf. [68]). Moreover, it is easy to construct them in such a way that the complete factorization of $p - 1$ is known, which makes it easy to find primitive roots g modulo p and prove that they are primitive.

The discrete logarithm problem in fields $GF(p)$ for which $p - 1$ does have a large prime factor appears to be about as hard as the problem of factoring integers of size about p. The comparison of the asymptotic complexity of the two problems was presented in Section 7. As far as values of practical use are concerned, the best current factorization programs appear to be capable of factoring integers around 2^{250} in about 1 day on a supercomputer like the Cray-XMP [22]. In applications where one is only interested in exchanging keys for use with ordinary cryptographic equipment, the Diffie-Hellman scheme presented in the Section 2 seems comparable to the Rivest-Shamir-Adleman (RSA) scheme, provided one uses fields $GF(p)$. However, the best choice is not totally obvious. The Diffie-Hellman scheme has the advantage that the parties exchanging keys do not have to keep their private keys secret (since there are no private keys). It has the disadvantage that there is no authentication. Furthermore, if the Diffie-Hellman scheme is to be used with the same field shared by many people for a prolonged time, the discrete logarithm problem being as hard as factorization loses some of its significance because the cryptanalyst can afford to spend much more time compiling the database. If the field to be used does change from one session to another, though, the Diffie-Hellman scheme appears as a good choice among key distribution systems.

Acknowledgement

The author thanks D. Coppersmith, J. Davenport, J. Davis, D. Ditzel, G. Gonnet, D. Holdridge, J. Jordan, N. K. Karmarkar, L. Kaufman, H. W. Lenstra, Jr., R. Mullin, C. Pomerance, J. Reyneri, B. Richmond, N. Sollenberger, J. Todd, S. Vanstone, and D. Wiedemann for helpful conversations.

Appendix A: Computation and estimation of $N(n,m)$, the number of polynomials over $GF(2)$ of degree n, all of whose irreducible factors are of degrees $\leq m$.

Let $I(k)$ denote the number of irreducible polynomials over $GF(2)$ that are of degree k. Then it is well-known [38] that

$$I(k) = \frac{1}{k} \sum_{d/k} \mu(d) 2^{k/d} , \tag{A.1}$$

where $\mu(d)$ is the Möbius μ-function. The formula (A.1) provides an efficient method for computing $I(k)$, the first few values of which are shown in Table 7. In addition, (A.1) shows immediately that

$$I(k) = k^{-1} 2^k + O(k^{-1} 2^{k/2}) . \tag{A.2}$$

We define $N(k,0) = 1$ if $k = 0$ and $N(k,0) = 0$ if $k \neq 0$. Also, we adopt the convention that $N(k,m) = 0$ if $k < 0$ and $m \geq 0$. With these conventions, we obtain the following recurrence, valid for n, $m > 0$:

$$N(n,m) = \sum_{k=1}^{m} \sum_{r \geq 1} N(n-rk, k-1) \binom{r + I(k) - 1}{r} . \tag{A.3}$$

To prove the validity of (A.3), note that any polynomial $f(x)$ of degree n, all of whose irreducible factors are of degrees $\leq m$, can be written uniquely as

$$f(x) = g(x) \prod_{u(x)} u(x)^{a(u(x))} ,$$

where the $u(x)$ are all of degree k for some k, $1 \leq k \leq m$, $\Sigma\, a(n(x)) = r$ for some $r \in Z^+$, and $g(x)$ is a polynomial of degree $n-rk$, all of whose irreducible factors are of degrees $\leq k-1$. Given k and r, there are $N(n-rk, k-1)$ such polynomials $g(x)$. The number of $\Pi\, u(x)^{a(n(x))}$ is the number of $I(k)$-tuples of nonnegative integers which sum to r, which is easily seen to equal

$$\binom{r + I(k) - 1}{r} .$$

This proves (A.3).

The recurrence (A.3) was used to compute the probabilities $p(n,m)$ listed in Appendix B. To estimate

Table 7. Values of $I(n)$, the number of irreducible binary polynomials of degree n.

n	$I(n)$
1	2
2	1
3	2
4	3
5	6
6	9
7	18
8	30
9	56
10	99
11	186
12	335
13	630
14	1161
15	2182
16	4080
17	7710
18	14532
19	27594
20	52377

$N(n,m)$ asymptotically, we use different techniques. The method we use differs from those used to study the analogous problem for ordinary integers (see [29,31,39,46]), and relies on the saddle point method [15]. With extra effort, it is capable of producing much more refined estimates than we obtain over much greater ranges of n and m. However, in order to keep the presentation simple, we consider only the ranges most important for cryptographic applications.

Theorem A1. Let

$$f_m(z) = \prod_{k=1}^{m} (1-z^k)^{-I(k)} \tag{A.4}$$

and

$$b(z) = \left[\frac{f'_m}{f_m}(z) \right]' = \sum_{k=1}^{m} I(k) \left\{ \frac{k(k-1)z^{k-2}}{1-z^k} + \frac{k^2 z^{2k-2}}{(1-z^k)^2} \right\} . \tag{A.5}$$

Then, for $n^{1/100} \le m \le n^{99/100}$, we have

$$N(n,m) \sim \left[2\pi b(r_0)\right]^{-1/2} f_m(r_0) r_0^{-n} \quad as \ n \to \infty, \tag{A.6}$$

where $r = r_0 = r_0(m,n)$ *is the unique positive solution to*

$$r \ \frac{f'_m}{f_m}(r) = n \ . \tag{A.7}$$

Corollary A2. If $n^{1/100} \le m \le n^{99/100}$, *then*

$$N(n,m) = 2^n \left[\frac{m}{n}\right]^{(1+o(1))n/m} \quad as \ n \to \infty. \tag{A.8}$$

Corollary A3. If $n^{1/100} \le m \le n^{99/100}$, and $0 \le k \le 2m$, *then*

$$\frac{N(n+k,m)}{N(n,m)} \sim 2^k \left[\frac{n}{m} \ \log_e \frac{n}{m}\right]^{k/m} \quad as \ n \to \infty \ . \tag{A.9}$$

Proof of Theorem A1. It is immediate from (A.4) that

$$f_m(z) = \prod_{k=1}^{m} \ (1+z^k+z^{2k}+\ldots)^{I(k)} = \sum_{n=0}^{\infty} N(n,m) z^n \ . \tag{A.10}$$

Hence, by Cauchy's theorem,

$$N(n,m) = \frac{1}{2\pi i} \int_{|z|=r} f_m(z) z^{-n-1} dz \tag{A.11}$$

$$= \frac{1}{2\pi} \int_{-\pi}^{\pi} f_m(re^{i\theta}) r^{-n} e^{-in\theta} d\theta \ ,$$

where r is any real number with $0 < r < 1$. As usual in the saddle point method, we determine $r = r_0$ by the condition (A.7), which is equivalent to

$$\sum_{k=1}^{m} I(k) \frac{kr^k}{1-r^k} = n \ . \tag{A.12}$$

Since $I(k) > 0$, it is clear that (A.12) has a unique solution $r = r_0$ with $0 < r_0 < 1$. We next estimate r_0 and $f_m(r_0)$. We consider $n^{1/100} \le m \le n^{99/100}$, $n \to \infty$, and take

$$r = \exp\left(\frac{\alpha \ \log \ n}{m} - \log \ 2\right), \qquad 10^{-3} \le \alpha \le 10^3 \ , \tag{A.13}$$

say. (All logrithms are to base e in this appendix.) Then, by (A.2),

$$\sum_{k=1}^{m} \frac{I(k)\, k r^k}{1-r^k} = \sum_{k=1}^{m} \frac{2^k r^k}{1-r^k} + O(1)$$

as $n \to \infty$ (uniformly in m satisfying $n^{1/100} \le m \le n^{99/100}$, as will be the case throughout the rest of the exposition), and so

$$\begin{aligned} \sum_{k=1}^{m} \frac{I(k)\, k r^k}{1-r^k} &= \sum_{k=1}^{m} 2^k r^k + O(1) \\ &= 2r\, \frac{2^m r^m - 1}{2r-1} + O(1) \qquad \text{(A.14)} \\ &= \frac{m(1+O(n^{-\alpha}+m^{-1}\log n))}{\alpha \log n}\, n^{\alpha} \ . \end{aligned}$$

We conclude that $r = r_0$ is given by (A.13) with $\alpha = \alpha_0$ satisfying

$$\alpha_0 = (\log n/m + \log\log n/m + o(1))\ (\log n)^{-1} \ , \qquad \text{(A.15)}$$

so that

$$\alpha_0 \sim \frac{\log n/m}{\log n} \quad \text{as} \quad n \to \infty \ , \qquad \text{(A.16)}$$

and that

$$2^m r_0^m \sim \alpha_0 n m^{-1} \log n \ . \qquad \text{(A.17)}$$

From (A.17) and (A.2) we easily conclude that

$$\sum_{k=1}^{m} - I(k) \log (1-r_0^k) = \sum_{k=1}^{m} k^{-1} 2^k r_0^k + O(1) \ , \qquad \text{(A.18)}$$

$$b(r_0) = \left(\frac{f_m'}{f_m}(r)\right)'\Big|_{r=r_0} = \sum_{k=1}^{m} \left\{ \frac{I(k)\, k(k-1)\, r_0^{k-2}}{1-r_0^k} + \frac{I(k)\, k^2 r_0^{2k-2}}{(1-r_0^k)^2} \right\} \sim 4\, mn \ , \qquad \text{(A.19)}$$

$$\left(\frac{f_m'}{f_m}(r)\right)''\Big|_{r=r_0} = O(m^2 n) \ . \qquad \text{(A.20)}$$

We now use the above estimates to carry out the saddle point approximation, which proceeds along very standard lines (cf. [15]). We choose

$$\theta_0 = m^{-1/2} n^{-599/1200} \ .$$

If we let

$$a(r_0) = \log f_m(r_0) - n \log r_0 ,$$

then by (A.12), (A.19), and (A.20) we obtain, for $|\theta| \le \theta_0$,

$$\log f_m(re^{i\theta}) - n \log r - in\theta = a(r_0) - \frac{1}{2} b(r_0)\theta^2 + O(m^2 n |\theta|^3)$$

$$= a(r_0) - \frac{1}{2} b(r_0)\theta^2 + O(m^{1/2} n^{-199/400}) .$$

Therefore

$$\frac{1}{2\pi} \int_{-\theta_0}^{\theta_0} f_m(r_0 e^{i\theta}) r_0^{-n} e^{-in\theta} d\theta$$

$$= (2\pi b(r_0))^{-1/2} f_m(r_0) r_0^{-n} (1 + O(m^{1/2} n^{-199/400})) .$$

It remains only to show that the integral over $\theta_0 < |\theta| \le \pi$ is negligible. Now for $z = re^{i\theta}$, $r = r_0$, and $m^* = [999m/1000]$, we have

$$\log f_m(r) - \log |f_m(z)| = \sum_{k=1}^{m} I(k) \log \left| \frac{1-z^k}{1-r^k} \right|$$

$$= \sum_{k=1}^{m} k^{-1} 2^k \{\log |1-z^k| - \log(1-r^k)\} + O(1)$$

$$= \sum_{k=1}^{m} k^{-1} 2^k r^k (1-\cos k\theta) + O(1)$$

$$\ge m^{-1} 2^{m^*} r^{m^*} \sum_{k=m^*}^{m} (1-\cos k\theta) + O(1) . \tag{A.21}$$

If $|\theta| \ge 10^4 m^{-1}$, say, the right side above is

$$= m^{-1} 2^{m^*} r^{m^*} \left[m - m^* + \frac{\sin (m + \frac{1}{2})\theta - \sin (m^* + \frac{1}{2})\theta}{2 \sin \theta/2} \right] + O(1)$$

$$\ge 10^{-4} 2^{m^*} r^{m^*} + O(1)$$

$$\ge 10^{-5} (n/m)^{999/1000} + O(1) . \tag{A.22}$$

If $\theta_0 \le |\theta| \le 10^4 m^{-1}$, on the other hand,

$$1 - \cos k\theta \geq 1 - \cos k\theta_0 \geq 10^{-3} mn^{-599/600} , \quad m^* \leq k \leq m ,$$

and the last quantity in (A.21) is

$$\geq 10^{-6} m^{1/1000} n^{1/1500} + O(1) . \tag{A.23}$$

Combining the estimates (A.22) and (A.23), we conclude that for $\theta_0 < |\theta| \leq \pi$,

$$\log f_m(r) - \log |f_m(z)| \geq \varepsilon n^{\delta}$$

for some $\varepsilon, \delta > 0$, and so the integral over that range is indeed negligible compared to the integral over $|\theta| \leq \theta_0$.

When we combine all the results obtained above, we obtain the estimate (A.6) of the theorem. *Proof of Corollary A2*. By (A.19), we know that $b(r_0) \sim 4mn$. Now

$$-n \log r_0 = n \log 2 - \frac{n}{m} \alpha_0 \log n \tag{A.24}$$

$$= n \log 2 - (1 + o(1)) \frac{n}{m} \log \frac{n}{m} .$$

Furthermore, by (A.17),

$$2^k r_0^k = O(n^{1/2} m^{-1/2} \log n) , \qquad k \leq m/2 ,$$

so

$$\sum_{k=1}^{m} k^{-1} 2^k r_0^k = O(n^{1/2} m^{-1/2} (\log n)^2) + O(nm^{-1}) ,$$

and so by (A.18),

$$\log f_m(r) = o(\frac{n}{m} \log \frac{n}{m}) .$$

This proves the estimate (A.8) of the corollary.

Proof of Corollary A3. We first study how $r_0 = r_0(n, m)$ varies with n. Letting n be a continuous variable defined by (A.12) (with m fixed and r varying), we find that (as in (A.19) and (A.20))

$$\left.\frac{\partial n}{\partial r}\right|_{r = r_0} \sim 2mn \ , \tag{A.25}$$

and for $|r - r_0| = O(n^{-1})$,

$$\frac{\partial^2 n}{\partial r^2} = O(m^2 n) \ .$$

Hence for $0 \le k \le 2m$,

$$\delta = r_0(n+k,\ m) - r_0(n,\ m) \sim k(2mn)^{-1} \ . \tag{A.26}$$

Therefore

$$\log\ f_m(r_0(n+k,\ m)) - \log\ f_m(r_0(n,\ m))$$

$$= \delta\ \frac{f_m'}{f_m}\ (r_0(r,\ k)) + O(\delta^2\ M) \ ,$$

where

$$M = \max_{r_0(n,\ k) \le r \le r_0(n,\ k)+\delta} \left| \left(\frac{f_m'}{f_m} \right)' \ (r) \right| = O(mn) \ ,$$

and so

$$\log\ f_m(r_0(n+k,\ m)) - \log\ f_m(r_0(n,\ m)) \sim k/m \ .$$

Since by (A.19),

$$b(r_0(n+k,\ m)) \sim b(r_0(n,\ m)) \sim 4mn \ ,$$

we finally obtain, for $r = r_0(n,\ m)$,

$$\frac{N(n+k,\ m)}{N(n,\ m)} \sim \exp\ (k/m)\ r^{-k}(1+\delta r^{-1})^{-n}$$

$$\sim r^{-k}$$

$$\sim 2^k \left(\frac{n}{m}\ \log\ \frac{n}{m} \right)^{-k/m} \ , \tag{A.27}$$

which yields the desired result.

Appendix B. Values of $p(n,k)$, the probability that a random polynomial over $GF(2)$ of degree n will have all its irreducible factors of degrees $\leq k$, for various values of n and for $1 \leq k \leq 40$.

$n = 10$

1	1.07422E-02	3.51562E-02	1.08398E-01	2.22656E-01	3.95508E-01
6	5.36133E-01	6.76758E-01	7.93945E-01	9.03320E-01	1.00000E+00
11	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
16	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
21	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
26	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
31	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
36	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00

$n = 20$

1	2.00272E-05	1.15395E-04	8.44955E-04	3.95012E-03	1.65253E-02
6	4.19769E-02	9.27200E-02	1.58895E-01	2.41888E-01	3.33941E-01
11	4.24762E-01	5.06549E-01	5.83453E-01	6.54315E-01	7.20904E-01
16	7.83160E-01	8.41983E-01	8.97418E-01	9.50049E-01	1.00000E+00
21	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
26	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
31	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
36	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00

$n = 30$

1	2.88710E-08	2.38419E-07	3.20468E-06	2.89446E-05	2.70738E-04
6	1.30629E-03	5.32556E-03	1.46109E-02	3.27337E-02	6.05388E-02
11	9.95504E-02	1.46035E-01	2.00105E-01	2.59328E-01	3.23701E-01
16	3.85957E-01	4.44780E-01	5.00215E-01	5.52846E-01	6.02797E-01
21	6.50413E-01	6.95845E-01	7.39323E-01	7.80979E-01	8.20979E-01
26	8.59436E-01	8.96473E-01	9.32185E-01	9.66668E-01	1.00000E+00
31	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
36	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00

$n = 40$

1	3.72893E-11	4.01087E-10	8.58199E-09	1.33864E-07	2.58580E-06
6	2.33114E-05	1.79979E-04	8.13273E-04	2.79863E-03	7.24926E-03
11	1.58528E-02	2.93316E-02	4.89490E-02	7.46204E-02	1.05880E-01
16	1.41606E-01	1.81373E-01	2.24427E-01	2.70539E-01	3.19242E-01
21	3.66858E-01	4.12290E-01	4.55768E-01	4.97424E-01	5.37424E-01
26	5.75881E-01	6.12918E-01	6.48630E-01	6.83113E-01	7.16445E-01
31	7.48703E-01	7.79953E-01	8.10256E-01	8.39667E-01	8.68239E-01
36	8.96016E-01	9.23043E-01	9.49359E-01	9.75000E-01	1.00000E+00

$n = 50$

1	4.52971E-14	6.00409E-13	1.87370E-11	4.64628E-10	1.72661E-08
6	2.83341E-07	4.15352E-06	3.15683E-05	1.70819E-04	6.37026E-04
11	1.89641E-03	4.51306E-03	9.32956E-03	1.69499E-02	2.80019E-02
16	4.27905E-02	6.17055E-02	8.43781E-02	1.10392E-01	1.39255E-01
21	1.70678E-01	2.04341E-01	2.40042E-01	2.77562E-01	3.16762E-01
26	3.55219E-01	3.92256E-01	4.27968E-01	4.62450E-01	4.95783E-01
31	5.28041E-01	5.59290E-01	5.89593E-01	6.19005E-01	6.47576E-01
36	6.75354E-01	7.02381E-01	7.28697E-01	7.54338E-01	7.79338E-01

$n = 60$

1	5.29091E-17	8.33535E-16	3.57405E-14	1.32500E-12	8.91426E-11
6	2.58718E-09	7.15988E-08	9.24769E-07	8.00984E-06	4.38229E-05
11	1.80902E-04	5.62442E-04	1.45972E-03	3.20781E-03	6.25179E-03
16	1.10167E-02	1.79110E-02	2.71835E-02	3.91011E-02	5.38207E-02
21	7.13458E-02	9.13458E-02	1.13563E-01	1.37747E-01	1.63715E-01
26	1.91288E-01	2.20330E-01	2.50714E-01	2.82341E-01	3.15118E-01
31	3.47376E-01	3.78625E-01	4.08928E-01	4.38340E-01	4.66911E-01
36	4.94689E-01	5.21716E-01	5.48032E-01	5.73673E-01	5.98673E-01

$n = 70$

1	6.01393E-20	1.09775E-18	6.19062E-17	3.27368E-15	3.78790E-13
6	1.88957E-11	9.76926E-10	2.15538E-08	3.02670E-07	2.46433E-06
11	1.43153E-05	5.88968E-05	1.94202E-04	5.22033E-04	1.21092E-03
16	2.47391E-03	4.57774E-03	7.79633E-03	1.24157E-02	1.86403E-02
21	2.66552E-02	3.65932E-02	4.85857E-02	6.26843E-02	7.87250E-02
26	9.65101E-02	1.15878E-01	1.36683E-01	1.58803E-01	1.82129E-01
31	2.06566E-01	2.32034E-01	2.58460E-01	2.85782E-01	3.13945E-01
36	3.41723E-01	3.68750E-01	3.95065E-01	4.20706E-01	4.45706E-01

$n = 80$

1	6.70016E-23	1.39049E-21	9.97836E-20	7.25067E-18	1.38122E-15
6	1.15256E-13	1.09894E-11	4.14853E-10	9.53139E-09	1.16815E-07
11	9.66482E-07	5.31900E-06	2.25093E-05	7.46852E-05	2.07615E-04
16	4.95625E-04	1.05199E-03	2.01760E-03	3.56457E-03	5.87520E-03
21	9.14845E-03	1.35494E-02	1.92187E-02	2.62653E-02	3.47883E-02
26	4.48677E-02	5.65725E-02	6.98314E-02	8.45073E-02	1.00477E-01
31	1.17636E-01	1.35888E-01	1.55151E-01	1.75351E-01	1.96424E-01
36	2.18311E-01	2.40962E-01	2.64330E-01	2.88376E-01	3.13064E-01

$n = 90$

1	7.35092E-26	1.70929E-24	1.52104E-22	1.47329E-20	4.45086E-18
6	6.05742E-16	1.05018E-13	6.77815E-12	2.56475E-10	4.77572E-09
11	5.68482E-08	4.22404E-07	2.31429E-06	9.54889E-06	3.20299E-05
16	8.99488E-05	2.19926E-04	4.77206E-04	9.41494E-04	1.71113E-03
21	2.90283E-03	4.64261E-03	7.06931E-03	1.03132E-02	1.44861E-02
26	1.96783E-02	2.59683E-02	3.34221E-02	4.20997E-02	5.20539E-02
31	6.32769E-02	7.56710E-02	8.91447E-02	1.03616E-01	1.19013E-01

36	1.35270E-01	1.52328E-01	1.70136E-01	1.88646E-01	2.07817E-01

$n = 100$

1	7.96750E-29	2.05183E-27	2.21718E-25	2.79208E-23	1.29509E-20
6	2.80812E-18	8.72341E-16	9.60294E-14	6.01224E-12	1.71395E-10
11	2.96075E-09	2.99434E-08	2.14020E-07	1.10537E-06	4.50209E-06
16	1.49474E-05	4.23006E-05	1.04387E-04	2.30656E-04	4.63358E-04
21	8.60536E-04	1.49335E-03	2.44477E-03	3.80487E-03	5.67219E-03
26	8.14723E-03	1.13206E-02	1.52676E-02	2.00530E-02	2.57320E-02
31	3.23535E-02	3.99608E-02	4.85939E-02	5.82726E-02	6.89360E-02
36	8.05132E-02	9.29404E-02	1.06160E-01	1.20121E-01	1.34775E-01

$n = 120$

1	9.10303E-35	2.79937E-33	4.24841E-31	8.51575E-29	8.58548E-26
6	4.43640E-23	4.23359E-20	1.33993E-17	2.31621E-15	1.57566E-13
11	5.85613E-12	1.12008E-10	1.38923E-09	1.14391E-08	6.97729E-08
16	3.28256E-07	1.26087E-06	4.06533E-06	1.13862E-05	2.82395E-05
21	6.33049E-05	1.30082E-04	2.48115E-04	4.43550E-04	7.50092E-04
26	1.20819E-03	1.86404E-03	2.76800E-03	3.97434E-03	5.54048E-03
31	7.52560E-03	9.98318E-03	1.29596E-02	1.64949E-02	2.06246E-02
36	2.53799E-02	3.07891E-02	3.68775E-02	4.36690E-02	5.11857E-02

$n = 140$

1	1.01163E-40	3.61674E-39	7.33734E-37	2.19965E-34	4.41316E-31
6	5.04962E-28	1.39814E-24	1.24688E-21	5.97886E-19	9.87955E-17
11	8.08426E-15	2.99212E-13	6.58221E-12	8.81292E-11	8.19347E-10
16	5.55000E-09	2.93431E-08	1.25224E-07	4.49650E-07	1.39106E-06
21	3.80057E-06	9.32833E-06	2.08939E-05	4.32132E-05	8.34419E-05
26	1.51678E-04	2.61437E-04	4.29945E-04	6.78498E-04	1.03218E-03
31	1.51926E-03	2.17048E-03	3.01880E-03	4.09904E-03	5.44782E-03
36	7.10258E-03	9.09777E-03	1.14637E-02	1.42271E-02	1.74120E-02

$n = 160$

1	1.10161E-46	4.48922E-45	1.17344E-42	5.01669E-40	1.86681E-36
6	4.44249E-33	3.38239E-29	8.30467E-26	1.10376E-22	4.48551E-20
11	8.22931E-18	6.00596E-16	2.38665E-14	5.28278E-13	7.59978E-12
16	7.51169E-11	5.53295E-10	3.15918E-09	1.46872E-08	5.71758E-08
21	1.91933E-07	5.66887E-07	1.50111E-06	3.61395E-06	8.01344E-06
26	1.65279E-05	3.19895E-05	5.85387E-05	1.01940E-04	1.69814E-04
31	2.71850E-04	4.19968E-04	6.28482E-04	9.13982E-04	1.29497E-03
36	1.79151E-03	2.42504E-03	3.21815E-03	4.19454E-03	5.37894E-03

$n = 180$

1	1.18108E-52	5.40360E-51	1.76869E-48	1.03860E-45	6.76840E-42
6	3.17410E-38	6.32280E-34	4.17015E-30	1.52942E-26	1.54164E-23

11	6.43456E-21	9.40765E-19	6.85942E-17	2.54621E-15	5.74207E-14
16	8.37861E-13	8.68865E-12	6.70002E-11	4.06745E-10	2.00783E-09
21	8.33948E-09	2.98308E-08	9.39352E-08	2.64665E-07	6.77278E-07
26	1.59242E-06	3.47604E-06	7.10546E-06	1.37023E-05	2.50828E-05
31	4.38309E-05	7.34728E-05	1.18633E-04	1.85151E-04	2.80190E-04
36	4.12329E-04	5.91651E-04	8.29679E-04	1.13916E-03	1.53389E-03

$n = 200$

1	1.25083E-58	6.34810E-57	2.54336E-54	1.99006E-51	2.16542E-47
6	1.90979E-43	9.50701E-39	1.64269E-34	1.65088E-30	4.15085E-27
11	3.98840E-24	1.18400E-21	1.60566E-19	1.01212E-17	3.61920E-16
16	7.87667E-15	1.16065E-13	1.21882E-12	9.73478E-12	6.13504E-11
21	3.17240E-10	1.38215E-09	5.20253E-09	1.72365E-08	5.11289E-08
26	1.37605E-07	3.40048E-07	7.79144E-07	1.66938E-06	3.36899E-06
31	6.44534E-06	1.17526E-05	2.05210E-05	3.44554E-05	5.58425E-05
36	8.76481E-05	1.33598E-04	1.98241E-04	2.87013E-04	4.06290E-04

$n = 250$

1	1.38731E-73	8.77490E-72	5.40876E-69	7.83967E-66	2.60927E-61
6	9.43320E-57	3.88248E-51	6.69001E-46	8.02116E-41	2.00422E-36
11	1.60958E-32	2.94386E-29	1.97886E-26	4.90886E-24	5.79891E-22
16	3.56248E-20	1.30783E-18	3.06590E-17	5.00010E-16	5.95292E-15
21	5.44802E-14	3.96906E-13	2.37961E-12	1.20333E-11	5.24971E-11
26	2.01111E-10	6.87289E-10	2.12261E-09	5.99202E-09	1.56105E-08
31	3.78515E-08	8.60400E-08	1.84519E-07	3.75441E-07	7.28331E-07
36	1.35288E-06	2.41546E-06	4.15966E-06	6.93046E-06	1.12015E-05

$n = 300$

1	1.47764E-88	1.11932E-86	9.83244E-84	2.37616E-80	2.02941E-75
6	2.48750E-70	6.82386E-64	1.01320E-57	1.35211E-51	3.35347E-46
11	2.33703E-41	2.77543E-37	9.78889E-34	1.00885E-30	4.14030E-28
16	7.51397E-26	7.16126E-24	3.88900E-22	1.33922E-20	3.10487E-19
21	5.17007E-18	6.45934E-17	6.31276E-16	4.97725E-15	3.25700E-14
26	1.80862E-13	8.69349E-13	3.67642E-12	1.38762E-11	4.73123E-11
31	1.47287E-10	4.22472E-10	1.12559E-09	2.80512E-09	6.57962E-09
36	1.46051E-08	3.08299E-08	6.21540E-08	1.20132E-07	2.23377E-07

$n = 350$

1	1.53041-103	1.35060-101	1.60284E-98	5.99444E-95	1.15598E-89
6	4.15490E-84	6.36291E-77	7.12431E-70	9.81396E-63	2.38010E-56
11	1.47173E-50	1.17990E-45	2.28345E-41	1.02142E-37	1.51735E-34
16	8.44461E-32	2.16145E-29	2.80349E-27	2.09567E-25	9.70161E-24
21	3.00716E-22	6.57878E-21	1.06830E-19	1.33657E-18	1.33338E-17
26	1.08953E-16	7.46950E-16	4.38196E-15	2.23833E-14	1.01023E-13
31	4.08092E-13	1.49196E-12	4.98491E-12	1.53517E-11	4.39066E-11
36	1.17397E-10	2.95185E-10	7.01651E-10	1.58406E-09	3.41082E-09

$n = 400$

1	1.55291-118	1.56457-116	2.41161-113	1.32035-109	5.21575-104
6	4.90260E-98	3.61544E-90	2.70641E-82	3.56518E-74	8.25910E-67
11	4.58566E-60	2.55583E-54	2.81416E-49	5.66577E-45	3.15379E-41
16	5.55533E-38	3.93027E-35	1.24956E-32	2.07597E-30	1.96036E-28
21	1.15333E-26	4.49726E-25	1.23335E-23	2.48551E-22	3.83276E-21
26	4.66781E-20	4.61886E-19	3.80070E-18	2.65470E-17	1.60138E-16
31	8.47073E-16	3.98086E-15	1.68143E-14	6.44748E-14	2.26456E-13
36	7.34282E-13	2.21339E-12	6.24107E-12	1.65527E-11	4.14991E-11

$n = 450$

1	1.55124-133	1.75679-131	3.41223-128	2.61980-124	1.96368-118
6	4.40044-112	1.37925-103	6.19409E-95	7.23803E-86	1.55644E-77
11	7.79521E-70	3.08892E-63	1.99379E-57	1.86331E-52	4.00361E-48
16	2.29399E-44	4.59937E-41	3.66621E-38	1.38170E-35	2.71137E-33
21	3.07933E-31	2.17355E-29	1.02106E-27	3.35795E-26	8.10087E-25
26	1.48691E-23	2.14569E-22	2.50052E-21	2.40980E-20	1.95927E-19
31	1.36783E-18	8.32458E-18	4.47600E-17	2.15115E-16	9.33645E-16
36	3.69290E-15	1.34195E-14	4.51235E-14	1.41305E-13	4.14464E-13

$n = 500$

1	1.53052-148	1.92464-146	4.59900-143	4.78471-139	6.39731-133
6	3.16656-126	3.79129-117	9.26531-108	8.93129E-98	1.72679E-88
11	7.79518E-80	2.23403E-72	8.66735E-66	3.86142E-60	3.28692E-55
16	6.27531E-51	3.64507E-47	7.43141E-44	6.46943E-41	2.68186E-38
21	5.96843E-36	7.73090E-34	6.29946E-32	3.42003E-30	1.30459E-28
26	3.64471E-27	7.74067E-26	1.28847E-24	1.72694E-23	1.90660E-22
31	1.76903E-21	1.40341E-20	9.66536E-20	5.85587E-19	3.15796E-18
36	1.53163E-17	6.74271E-17	2.71643E-16	1.00884E-15	3.47656E-15

References

1. L. M. Adleman, A subexponential algorithm for the discrete logarithm problem with applications to cryptography, *Proc. 20th IEEE Found. Comp. Sci. Symp.* (1979), 55-60.

2. L. M. Adleman, C. Pomerance and R. S. Rumely, On distinguishing prime numbers from composite numbers, *Annals Math. 117* (1983), 173-206.

3. B. Arazi, Sequences constructed by operations modulo $2^n - 1$ or modulo 2^n and their application in evaluating the complexity of a log operation over $GF(2^n)$, preprint.

4. C. P. Arnold, M. I. Parr, and M. B. Dewe, An efficient parallel algorithm for the solution of large sparse linear matrix equations, *IEEE Trans. on Computers, C-32* (1983), 265-272.

5. E. Bach, Discrete logarithms and factoring, to be published.

6. V. A. Barker, ed., *Sparse Matrix Techniques*, Lecture Notes in Mathematics #572, Springer-Verlag, 1977.

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

8. I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone, Computing logarithms in finite fields of characteristic two, *SIAM J. Alg. Disc. Methods, 5* (1984), 276-285.

9. M. Blum and S. Micali, How to generate cryptographically strong sequences of pseudo random bits, *SIAM J. Comp.*, to appear.

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

11. A. Brameller, R. N. Allan, and Y. M. Hamam, *Sparsity*, Pitman 1976.

12. E. F. Brickell, A fast modular multiplication algorithm with applications to two key crytography, pp. 51-60 in *Advances in Cryptology: Proceedings of CRYPTO '82*, D. Chaum, R. Rivest, and A. Sherman, eds., Plenum Press, 1983.

13. E. F. Brickell and J. H. Moore, Some remarks on the Herlestam-Johannesson algorithm for computing logarithms over $GF(2^n)$, pp. 15-20, in *Advances in Cryptology: Proceedings of CRYPTO '82*, D. Chaum, R. Rivest and A. Sherman, eds., Plenum Press, 1983.

14. J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, and S. S. Wagstaff, Jr., *Factorizations of* $b^n \pm 1, b = 2, 3, 5, 6, 7, 10, 11, 12$ *up to High Powers*, Am. Math. Society, 1983.

15. N. G. de Bruijn, *Asymptotic Methods in Analysis*, North-Holland. 1958,

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

17. H. Cohen and H. W. Lenstra, Jr., Primality testing and Jacobi sums, *Math. Comp., 42* (1984), 297-330.

18. D. Coppersmith, Evaluating logarithms in $GF(2^n)$, pp. 201-207 in *Proc. 16th ACM Symp. Theory of Computing*, 1984.

19. D. Coppersmith, Fast evaluation of logarithms in fields of characteristic two, *IEEE Trans. Inform. Theory* IT-30 (1984), 587-594.

20. D. Coppersmith and A. M. Odlyzko, manuscript in preparation.

21. D. Coppersmith and S. Winograd, On the asymptotic complexity of matrix multiplication, *SIAM J. Comp. 11* (1982), 472-492.

22. J. A. Davis, D. B. Holdridge, and G. J. Simmons, Status report on factoring (at the Sandia National Laboratories), to appear in Proc. EUROCRYPT 84.

23. W. Diffie and M. E. Hellman, New directions in cryptography, *IEEE Trans. Inform. Theory*, IT-22 (1976), 644-654.

24. W. Diffie and M. E. Hellman, Exhaustive cryptanalysis of the NBS Data Encryption Standard, *Computer 10* (1977), 74-84.

25. T. ElGamal, A subexponential-time algorithm for computing discrete logarithms over $GF(p^2)$, *IEEE Trans. Inform. Theory*, to appear.

26. T. ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. Inform. Theory*, to appear.

27. A. George and J. W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems,* Prentice-Hall, 1981.

28. S. Golomb, *Shift-register Sequences,* Holden-Day, 1967.

29. F. G. Gustavson, Analysis of the Berlekamp-Massey feedback shift-register synthesis algorithm, *IBM J. Res. Dev. 20* (1976), 204-212.

30. M. E. Hellman and J. M. Reyneri, Fast computation of discrete logarithms in $GF(q)$, pp. 3-13 in *Advances in Cryptography: Proceedings of CRYPTO '82,* D. Chaum, R. Rivest, and A. Sherman, eds., Plenum Press, 1983.

31. D. Hensley, The number of positive integers $\leq x$ and free of prime factors $> y$, preprint.

32. T. Herlestam and R. Johannesson, On computing logarithms over $GF(2^p)$, *BIT 21* (1981), 326-334.

33. M. R. Hestenes and E. Stiefel, Methods of conjugate gradients for solving linear systems, *J. Res. Nat. Bureau of Standards 49* (1952), 409-436.

34. A. Hildebrand, On the number of positive integers $\leq x$ and free of prime factors $> y$, to be published.

35. J. Ja' Ja' and S. Venkatesan, On the complexity of a parity problem related to coding theory, Pennsylvania State Univ. Computer Sci. Report CS-81-5 (1981).

36. D. E. Knuth, *The Art of Computer Programming: Vol. 2,* Seminumerical Algorithms, 2nd ed., Addison-Wesley 1981.

37. A. G. Konheim, *Cryptography: A Primer*, Wiley, 19981.

38. J. Kowalchuk, B. P. Schanning, and S. Powers, Communication privacy: Integration of public and secret key cryptography, *NTC Conference Record*, Vol. 3, pp. 49.1.1-49.1.5, Dec. 1980.

39. C. Lanczos, Solution of systems of linear equations by minimized iterations, *J. Res. Nat. Bureau of Standards 49* (1952), 33-53.

40. D. L. Long, Random equivalence of factorization and computation of orders, *Theoretical Comp. Sci.*, to appear.

41. D. L. Long and A. Wigderson, How discreet is the discrete log?, pp. 413-420 in *Proc. 15-th ACM Symp. Theory of Computing*, 1983.

42. F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland, 1977.

43. H. Maier, On integers free of large prime divisors, to be published.

44. J. L. Massey, Shift-register synthesis and BCH decoding, *IEEE Trans. Inform. Theory IT-15* (1969), 122-127.

45. J. L. Massey, Logarithms in finite cyclic groups - cryptographic issues, pp. 17-25 in *Proc. 4th Benelux Symp. on Inform. Theory*, Leuven, Belgium, May 1983.

46. R. Merkle, Secrecy, authentication, and public key systems, Ph.D. dissertation, Dept. of Electrical Engineering, Stanford Univ., 1979.

47. J. C. P. Miller, On factorization, with a suggested new approach, *Math. Comp. 29* (1975), 155-172.

48. R. C. Mullin and S. A. Vanstone, manuscript in preparation.

49. R. W. K. Odoni, V. Varadharajan, and P. W. Sanders, Public key distribution in matrix rings, *Electronics Letters 20* (1984), 386-387.

50. H. Ong, C. P. Schnorr, and A. Shamir, An efficient signature scheme based on quadratic forms, pp. 208-216 in *Proc. 16th ACM Symp. Theory of Comp.*, 1984.

51. S. C. Pohlig and M. Hellman, An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance, *IEEE Trans. Inform. Theory* IT-24 (1978), 106-110.

52. J. Pollard, Monte Carlo methods for index computations $(\mathrm{mod}\ p)$, *Math. Comp. 32* (1978), 918-924.

53. C. Pomerance, Analysis and comparison of some integer factoring algorithms, pp. 89-139 in *Computational Methods in Number Theory: Part 1,* H. W. Lenstra, Jr., and R. Tijdeman, eds., Math. Centre Tract 154, Math. Centre Amsterdam, 1982.

54. G. B. Purdy, A high security log-in procedure, *Comm. ACM 17* (1974), 442-445.

55. M. O. Rabin, Probabilistic algorithms in finite fields, *SIAM J. Comp. 9* (1980), 273-280.

56. J. A. Reeds and N. J. A. Sloane, Shift-register synthesis (modulo *m*), *SIAM J. Comp.,* to appear.

57. J. E. Sachs and S. Berkovits, Probabilistic analysis and performance modelling of the "Swedish" algorithm and modifications, to be published.

58. J. Sattler and C. P. Schnorr, Generating random walks in groups, preprint.

59. B. P. Schanning, Data encryption with public key distribution, *EASCON Conf. Rec.*, Washington, D.C., Oct. 1979, pp. 653-660.

60. C. P. Schnorr and H. W. Lenstra, Jr., A Monte Carlo factoring algorithm with linear storage, *Math. Comp. 43* (1984), 289-311.

61. R. Schreiber, A new implementation of sparse gaussian elimination, *ACM Trans. Math. Software 8* (1982), 256-276.

62. J. W. Smith and S. S. Wagstaff, Jr., An extended precision operand computer, pp. 209-216 in *Proc. 21st Southeast Region. ACM Conference*, 1983.

63. P. K. S. Wah and M. Z. Wang, Realization and application of the Massey-Omura lock, pp. 175-182 in *Proc. Intern. Zurich Seminar*, March 6-8, 1984.

64. A. L. Wells, Jr., A polynomial form for logarithms modulo a prime, *IEEE Trans. Inform. Theory*, to appear.

65. A. E. Western and J. C. P. Miller, *Tables of Indices and Primitive Roots,* Royal Society Mathematical Tables, vol. 9, Cambridge Univ. Press, 1968.

66. D. Wiedemann, Solving sparse linear equations over finite fields, manuscript in preparation.

67. R. M. Willett, Finding logarithms over large finite fields, in prepration.

68. H. C. Williams and B. Schmid, Some remarks concerning the M.I.T. public-key system, *BIT 19* (1979), 525-538.

69. K. Yiu and K. Peterson, A single-chip VLSI implementation of the discrete exponential public key distribution system, *Proc. GLOBCOM-82*, IEEE 1982, pp. 173-179.

70. N. Zierler, A conversion algorithm for logarithms on $GF(2^n)$, *J. Pure Appl. Algebra 4* (1974), 353-356.

71. N. Zierler and J. Brillhart, On primitive trinomials (mod 2), *Inform. Control 13* (1968), 541-554.

72. N. Zierler and J. Brillhart, On primitive trinomials (mod 2), II., *Inform. Control 14* (1969), 566-569.