Proceedings of Symposia in Applied Mathematics
Volume 42, 1990

The Discrete Logarithm Problem

KEVIN S. McCURLEY

ABSTRACT. There are numerous cryptosystems whose security is based on the difficulty of solving the discrete logarithm problem. This paper is a survey of the discrete logarithm problem, including the current state of algorithms for solving it, complexity issues related to the discrete logarithm problem, and applications to cryptography.

If the theory of numbers could be employed for any practical and obviously honorable purpose ... then surely neither Gauss nor any other mathematician would have been so foolish as to decry or regret such applications.

— G. H. Hardy, *A Mathematician's Apology*, 1940

1. Introduction

We begin with a succinct statement of the computational problem that is the object of study in this paper. Let G be a group (written multiplicatively), and for $g \in G$, let $\langle g \rangle$ be the cyclic subgroup generated by g. The *discrete logarithm problem* for the group G may be stated as:

> Given $g \in G$ and $a \in \langle g \rangle$, find an integer x such that $g^x = a$.

Such an integer x is the discrete logarithm of a to the base g, and we shall use the notation $x = \operatorname{ind}_g a$ (another word for discrete logarithm is *index*). Note that $\operatorname{ind}_g a$ is only determined modulo the order of g. There are some subtleties to the precise statement of the problem, which we shall return to later in §3.

This paper is not intended as a completely comprehensive treatise on the discrete logarithm problem. Rather, it is intended to serve as an introduction to the subject for those merely interested in learning about applications of

1980 *Mathematics Subject Classification* (1985 *Revision*). Primary 11Y16, 11T71, 11A07.

A preliminary version of this paper was written while the author was at IBM Almaden Research Center, San Jose, California.

number theory, and for those seeking research problems in computational number theory. In this paper we shall cover some algorithms for solving the discrete logarithm problem, some complexity issues related to the problem, and some applications to cryptography.

At this time I would like to offer my thanks to Andrew Odlyzko and Carl Pomerance for suggesting corrections and improvements to an earlier version of this paper. I would also like to thank Jim Hafner for some helpful and interesting conversations during the writing of this paper.

2. Applications to Cryptography

The origin of using the discrete logarithm problem in cryptographic schemes goes back to the seminal paper of Diffie and Hellman [**21**]. It is there that they proposed the discrete logarithm problem as a good source for a "one-way function." A formal definition of a one-way function is beyond the scope of this paper (and not universally agreed upon!). Informally we may think of a one-way function as a function $f : X \to Y$ for which it is easy to compute $f(x)$ given $x \in X$, but given $y \in Y$, it is difficult to compute a value x with $f(x) = y$, at least for most values of y.

In the paper of Diffie and Hellman, they proposed as a natural candidate a function based on exponentiation modulo a prime. Let p be a large prime, and let g be a primitive root modulo p, i.e., a generator of the multiplicative group $\mathrm{GF}(p)^*$. We can then define a function $f : \{1, \ldots, p-1\} \to \mathrm{GF}(p)^*$ by $f(x) = g^x \pmod p$. It is easy to see that this function is relatively easy to compute, by using the binary expansion of x. Let $x = \sum_{i=0}^{k} \epsilon_i 2^i$ with $\epsilon_i = 0$ or 1. Then

$$g^x \equiv \prod_{\epsilon_i = 1} g^{2^i} \pmod p.$$

By repeated squaring and multiplication, we can easily compute the right side using at most $2k$ multiplications modulo p. (See [**32**, §4.6.3].) On the other hand, inverting the function f clearly requires an algorithm for the discrete logarithm problem in $\mathrm{GF}(p)^*$, and this is widely thought to be intractable for large p.

To see how such a one-way function can be used in cryptography, we give two examples. Apparently the first application of one-way functions in cryptography is due to Needham (see [**65**, p. 91]), who used them to devise a secure method for storing passwords in multiuser computer systems. Most such systems incorporate into their operating systems a method for authenticating users, usually through a password that is known to the user. If the computer system stores the passwords of all the users in a file, then this file must be heavily protected. In particular, the system resources can easily be compromised by an unscrupulous systems programmer. One way around this is to store not the passwords themselves, but rather the values of a one-way function applied to the passwords. When the user types in the password, the function is applied to the password and compared to the value stored in

the file, with login allowed if the values match. The file itself is, however, quite useless to an intruder. One-way functions are in fact used in a manner similar to this in both the UNIX[1] and VMS[2] operating systems.

The second and possibly the most important example of a cryptosystem based on the difficulty of computing discrete logarithms is due to Diffie and Hellman [**21**]. This is a method for two users to agree on a secret cryptographic key (string of bits) using only an insecure channel of communication. Before the discovery of public key systems, traditional cryptosystems required that the two parties wishing to communicate meet beforehand to agree upon a secret key. This severely limits the spontaneity of secure communication, and may require a courier. The Diffie-Hellman key selection protocol eliminates this problem. To describe it, we assume that two users, A and B, wish to agree upon a key. Through any means of communication, they first choose a group G and an element $g \in G$. There are several requirements that the group should satisfy for security reasons, but to begin with, it should have the property that it is easy to compute the product of any two elements in the group, and there should be an easy way of identifying group elements with keys. In the original Diffie-Hellman paper, they used $\mathrm{GF}(p)^*$ for a large prime p, so that there is a natural way of identifying group elements as positive integers.

The key construction between the two parties A and B (Anna and Boris?) proceeds as follows. Let M be a large integer (say $> 10^{40}$).

- A chooses a random integer x with $0 < x < M$, computes g^x, and sends the result to B, keeping x secret.
- B chooses a random integer y with $0 < y < M$, computes g^y, and sends the result to A, keeping y secret.
- Both A and B construct the key from g^{xy}, which A computes from $(g^y)^x$, and B computes from $(g^x)^y$.

If this scheme is to be secure, then the problem of computing g^{xy} from knowledge of g, g^x, and g^y should be intractable. We shall refer to this problem as the *Diffie-Hellman problem.* Clearly, if one can solve a generic instance of the discrete logarithm problem, then one can also solve the Diffie-Hellman problem. It remains an open question whether the converse is true (but see §3 for further discussion).

The Diffie-Hellman scheme has found widespread use in practical cryptosystems, as for example in the optional security features of the NFS file system of the SunOS[3] operating system. One of the often-cited disadvantages of public key systems is that they require significantly more computational power than traditional private key systems such as the Data Encryption Standard (known as DES). The Diffie-Hellman scheme has the nice property that

[1]UNIX is a trademark of AT&T Bell Laboratories.

[2]VMS is a trademark of Digital Equipment Corporation.

[3]NFS and SunOS are trademarks of Sun Microsystems, Inc.

a very fast scheme such as DES can be used for the actual encryption, yet it still enjoys one of the main advantages of public key cryptography.

It should be mentioned that there are numerous other cryptosystems that are based on the difficulty of the discrete logarithm problem, including those described in [6], [8], [11], [14], [15], [17], [22], [27], [30], [34], [37], [47], [52], and [58]. Yet another example is also given in the accompanying introductory paper by Carl Pomerance.

3. Computational Complexity Issues

If we wish to discuss the security of cryptosystems against computational attacks, then we are naturally led to a discussion of what is a feasible computation. The field of mathematics that deals with this is known as computational complexity theory, and the connections to cryptography are described in more detail in the accompanying paper by Shafi Goldwasser. In this section we shall briefly survey some complexity issues involving the discrete logarithm problem.

One of the first things that deserves attention is the statement of the problem itself. First of all, it should be noted that it makes sense to talk about the discrete logarithm problem in an arbitrary semigroup, although in most applications the interest is in specific examples of finite cyclic groups.

In the statement of the problem given in §1, we specifically excluded the possibility that a might not be in the cyclic group generated by g. An alternative way to state the problem is:

> Given a, $g \in G$, determine if there exists an integer x such that $g^x = a$, and if so, find such an x.

Note that this formulation of the problem might be harder to solve. It can also make a difference if we are analyzing an algorithm for solving the discrete logarithm problem, since we may need to know that $a \in \langle g \rangle$ in order to carry out the analysis. Henceforth we shall assume that our group G is cyclic, with $G = \langle g \rangle$, so that we may use the original statement of the discrete logarithm problem.

We should also be careful to make clear exactly what we mean when we say that we are "given the group G." Groups may arise in quite a few different abstract forms, for example, as the class group of an imaginary quadratic field, given by the discriminant, or the group of points on an elliptic curve over a finite field (or $\mathbb{Q}$), given by the equation of the curve and the specification of the field, or the Galois group of a polynomial, or a finite abelian group given by generators and relations, or a finite abelian group given by invariant factors. Rather than focus on such abstract issues, let us make the following assumptions about our group G:

- We know efficient algorithms for testing equality of group elements.
- We know an efficient algorithm for multiplying any two elements of the group.

- The group is finite, generated by g, with known order n.

The first assumption is intended to deal with the case where the group is given as a set of equivalence classes (as in a class group), since it may be quite a complex problem to decide if two objects belong to the same equivalence class. The need for the second assumption is fairly obvious, since we need an efficient algorithm for multiplying in the group if we are to use exponentiation in the group as the source of a one-way function.

It is probably less obvious why we should assume that the order of the element g is known. The reason for this is in fact rather simple, namely that any algorithm for computing discrete logarithms can be used to calculate the order of the base element g. To see why, assume that we are in possession of an algorithm to compute discrete logarithms to the base g, and for simplicity assume that the algorithm returns the logarithm from the interval $[1, n]$, where n is the order of g. To compute n, we first choose an integer m that we think exceeds n, or if we have no idea we simply choose $m = 1$. We then choose a random integer $y_0 \in \{m, \ldots, 2m\}$, and compute $a = g^{y_0}$. Next we use the discrete logarithm algorithm to compute $x_0 = \operatorname{ind}_g a$. It follows that $x_0 - y_0$ is a multiple of the order of g, and we initially set $n_0 = x_0 - y_0$. If m was initially chosen less than n, then n_0 will be zero, but this can be remedied by simply doubling m until a nonzero one is found. Note that since y_0 was randomly chosen, it follows that n_0 will be nearly uniformly distributed among the multiples of n in the interval $[m-n, 2m+n]$. We repeat the process to compute another random multiple n_1, and set $n_0 = \gcd(n_0, n_1)$. By taking the gcd of several random multiples of the order of g, we will very probably produce the order itself. Further discussion of this may be found in [**4**] and [**46**].

The field of computational complexity has as its goal the classification of computational problems according to their difficulty. The fact that a discrete logarithm algorithm can be used to compute the order of the base for the logarithms is an example of a random polynomial time reduction between two computational problems. Such reductions may be viewed as inducing a partial ordering on the set of computational problems according to their complexity, and can provide useful information on the intrinsic difficulty of a problem. In this case, we find that it is at least as difficult to compute discrete logarithms to a given base as it is to compute the order of that base in the group.

An interesting consequence of the preceding discussion is that it may be advantageous to choose a cyclic group for which the problem of computing the order is thought to be very difficult. For example, it is possible to design a Diffie-Hellman key selection system for which factorization of a large integer is required to break the system. The details of this may be found in [**46**], but the basic idea is to choose a subgroup of $(\mathbb{Z}/n\mathbb{Z})^*$ with $n = pq$ being a product of two large primes of a special form. Using this choice, it is possible

to prove that breaking the system requires the ability to factor n as well as break the scheme in the groups $\mathrm{GF}(p)^*$ and $\mathrm{GF}(q)^*$. Hence the system will remain secure if either of two presumably hard problems is in fact difficult to solve. A similar property is enjoyed by the system described in [**14**], where the cryptanalysis of the system requires computing the class number of an imaginary quadratic number field.

We should keep in mind that breaking the Diffie-Hellman key selection scheme may not in fact require the computation of any discrete logarithms. In §2, we pointed out that breaking the Diffie-Hellman scheme requires a solution to the Diffie-Hellman problem:

$$\text{Given } a, b, \text{ and } g \in G, \text{ find } g^{xy}, \text{ where } a = g^x \text{ and } b = g^y.$$

Clearly, if we can compute either x or y, then we can compute g^{xy} and thereby solve the Diffie-Hellman problem. It is not clear whether the reverse implication is true, namely whether an algorithm for solving the Diffie-Hellman problem can be used to compute discrete logarithms. Bert den Boer [**9**] recently proved that if $\varphi(p-1)$ has all small prime factors, then the two problems are in some sense equivalent for the group $\mathrm{GF}(p)^*$. The general problem remains open, however, and it might be the case for some groups that the Diffie-Hellman problem is easier because there is an efficient algorithm for solving it yet to be discovered. A similar situation exists for the RSA cryptosystem, in which the decryption of an individual cyphertext c requires the ability to solve the congruence $x^e \equiv c \pmod{n}$ for x when given c, n, and e satisfying $\gcd(e, \varphi(n)) = 1$. At present there is no proof that the ability to solve the congruence requires the ability to factor n.

Another point that we should take note of is the fact that if we can solve the discrete logarithm problem using one base g, then we can also solve the discrete logarithm problem for the base h, provided h belongs to the cyclic subgroup generated by g. This can easily be seen from the fact that if $a \in \langle h \rangle$, then

$$\operatorname{ind}_g a \equiv \operatorname{ind}_g h \cdot \operatorname{ind}_h a \pmod{n},$$

where n is the order of g.

Yet another interesting topic concerning complexity and the discrete logarithm problem concerns the security of individual bits of the discrete logarithm. The motivation for this investigation is the fact that while it may be difficult to calculate the entire discrete logarithm, it may be very easy to determine some of the bits of the discrete logarithm. On the other hand, for applications to cryptography, we would ideally like to prevent the cryptanalyst from discovering *any* partial information.

In fact it is very easy to give an example where some of the bits of the discrete logarithm are easy to find. If n is even, then it is easy to see that $\operatorname{ind}_g a$ is even if and only if a is a square. Moreover, we can easily determine if a is a square in the group since a is a square if and only if $a^{n/2} = 1_G$ (for $G = (\mathbb{Z}/p\mathbb{Z})^*$ we can also use quadratic reciprocity). Hence we can easily

determine the least significant bit of $\operatorname{ind}_g a$ in the case that n is even. In §4.2 we shall describe a complete discrete logarithm algorithm based on this approach.

Given that some of the bits can be easy to calculate, the question now arises as to which of the bits of the discrete logarithm are the most difficult to calculate, or perhaps more naturally how to extract secure bits from the discrete logarithm. Research in this area has been carried out by Blum and Micali [8], Peralta [53], and Long and Wigderson [43]. In their seminal paper, Blum and Micali defined the notion of a "hard predicate" for a one-way function, and showed the existence of such a predicate for exponentiation modulo a prime. Let $f : S \longrightarrow S$ be a one-way function. Then a Boolean predicate $B : S \longrightarrow \{0, 1\}$ is said to be hard for f if an oracle for $B(f(x))$ allows one to invert f easily (the oracle may even be probabilistic with only a slight advantage in predicting $B(f(x))$).

For a prime p and primitive root g, define $B : (\mathbb{Z}/p\mathbb{Z})^* \to \{0, 1\}$ by

$$B(a) = \begin{cases} 0, & \text{if } 1 \le \operatorname{ind}_g a \le (p-1)/2 \\ 1, & \text{if } (p-1)/2 < \operatorname{ind}_g a \le p-1. \end{cases}$$

It is rather easy to verify that an oracle for $B(\cdot)$ can be used to construct a simple algorithm for computing discrete logarithms. Blum and Micali [8] went even further to show that any oracle with a nonnegligible advantage in predicting $B(\cdot)$ can also be used to compute discrete logarithms. In this way they were able to prove that predicting this single bit from the discrete logarithm is essentially as difficult as predicting the entire discrete logarithm.

Peralta [53] and Long and Wigderson [43] later independently proved that there exists a constant c with the property that one can extract at least $c \log\log p$ bits from the discrete logarithm that are simultaneously secure (see the original papers for details). These results remain the best that are known, and it is an interesting open question whether one can extract $c \log p$ simultaneously secure bits for some constant c. Schrift and Shamir [60] recently took a step in this direction, but their results show a relationship between factoring a composite and computing individual bits of the discrete logarithm modulo the composite.

I have not completely exhausted the subject of computational complexity as it relates to the discrete logarithm problem. Some further interesting topics involve connections to other problems, such as the word problem in group theory, zero knowledge proofs [6], [17], [43], and others [10]. Further discussion on the complexity of other number theoretic problems can be found in [2], [5], and [24].

4. Algorithms for the Discrete Logarithm Problem

In any discussion of algorithms for computing discrete logarithms, we should make a distinction between algorithms that are designed to be practical and those that are structured in such a way as to allow a rigorous proof of their behavior. Consider for a moment the case of computing discrete

logarithms over $GF(2^k)^*$. For this problem, the algorithm with the fastest asymptotic running time is due to Coppersmith [18], but this is based on a heuristic analysis, and it remains an open problem to rigorously analyze its running time. The algorithm that has the asymptotically fastest rigorously proved running time is due to Pomerance [56], but this algorithm probably will not be competitive in practice.

It is also possible that the algorithm with the fastest asymptotic running time estimate may not be an ideal choice for implementation. In the design of practical algorithms, there are many issues involving hardware and software that will affect which algorithm will work the fastest. When a person is designing an algorithm that is actually to be implemented, a 50 percent improvement in the running time of the algorithm is sometimes significant, but if the goal is to prove a theorem of the form "the running time is $O(n^3)$ for an input n," then a 50 percent improvement in the running time will disappear into the implied constant and is of no value.

All of the known algorithms for solving the discrete logarithm problem may be placed into one of three categories. First are the algorithms that work for arbitrary groups, that is, those that do not exploit any specific property of the group. Next we have algorithms that work well in finite groups for which the order of the group has no large prime factors. Finally we have algorithms that exploit methods for representing group elements as products of elements from a relatively small set. In the subsections that follow we shall describe the basic ideas behind each of these methods. The emphasis will not be on giving the absolute best versions of each type of algorithm, but rather to illustrate the ideas involved and give further references. For simplicity we shall assume throughout this section that $g \in G$ is the base for which we wish to compute discrete logarithms and that g has known order n in G.

4.1. Algorithms for general groups. In this section we shall describe algorithms that work in the absence of any extra information concerning the group. The most obvious algorithm is simply to build a table containing all n powers of g and to simply look up group elements in the table to find their discrete logarithms. This evidently takes at least n operations to compute the table, and $O(n)$ space to store the table. The goal of any algorithm is to improve on these bounds.

The first improvement on the most obvious algorithm reduces the running time to $O(n^{1/2} \log n)$ and the space requirement to $O(n^{1/2})$ group elements. The method is attributed to Daniel Shanks and is described in [33, pp. 9, 575–576]. For this algorithm we shall require an enumeration of the elements of G, or more specifically an easily computable injective function $f : G \longrightarrow \{1, \ldots, n\}$. (Such a function might also be used in the Diffie-Hellman key selection procedure to construct keys from group elements.) For every $a \in G$, we may take $\operatorname{ind}_g a$ as an integer in the interval $[1, n]$, which means

that $\operatorname{ind}_g a$ has a decomposition in the form $mq - r$, where $m = \lceil n^{1/2} \rceil$, $0 \leq r < m$, and $0 \leq q \leq m$. In order to compute $\operatorname{ind}_g a$, it suffices to compute r and q, and we can do this as follows. We first compute the sets

$$S = \{(i, f(ag^i)),\ i = 0, \ldots, m\}$$
$$T = \{(i, f(g^{mi})),\ i = 0, \ldots, m\}.$$

We now sort both sets according to the second entries of the elements and run through the sorted lists to find elements $s \in S$ and $t \in T$ that agree in the second coordinate. From the first entries of these two elements we get $ag^r = g^{mq}$, so $a = g^{mq-r}$.

Using well-known sorting algorithms, we can sort the sets S and T in $O(m \log m)$ operations (see [**33**, Ch. 5] or [**3**, Ch. 3]). Hence this gives an algorithm for computing discrete logarithms that uses $O(n^{1/2} \log n)$ time and storage space for $O(n^{1/2})$ group elements.

Interestingly enough, Shanks' algorithm will work even if n is only an upper bound for the order of the group element g. We can even do away with the need for the upper bound since we can simply pick an integer n to use, and if the bound turns out to be too small, then we simply keep doubling it and repeating the steps until the answer is found. It therefore follows from the discussion of §3 that Shanks' algorithm can also be used to calculate the order of g in time $O(n^{1/2} \log n)$. (This is in fact the purpose for which Shanks originally presented the idea in [**61**].)

Shanks' algorithm should be regarded as being mostly of theoretical interest, since there is a very practical algorithm due to Pollard [**55**] that seems to have essentially the same running time and a much smaller space requirement. We pay for this improvement in two ways. First, the algorithm really does require the order of the group rather than just an upper bound, and second, we give up our rigorous proof of the running-time bound. There is, however, still a heuristic argument to support the conjectured running-time estimate.

In the first stage of the algorithm, we compute integers s and t such that

$$a^s = g^t. \tag{4.1}$$

The procedure for doing this will involve the construction of a sequence of group elements $x_0, x_1, \ldots$. We begin by partitioning the group G into three roughly equal-sized sets S_1, S_2, and S_3. Define $x_0 = 1_G$ and

$$x_{i+1} = \begin{cases} ax_i & \text{for } x_i \in S_1, \\ x_i^2 & \text{for } x_i \in S_2, \\ gx_i & \text{for } x_i \in S_3. \end{cases} \tag{4.2}$$

This defines a sequence of integers a_i and b_i for which $x_i = a^{a_i} g^{b_i}$. The rules for calculating a_i and b_i are easy to deduce from (4.2), namely that

$a_0 = b_0 = 0$,

$$a_{i+1} = \begin{cases} a_i + 1 \pmod n & \text{for } x_i \in S_1, \\ 2a_i \pmod n & \text{for } x_i \in S_2, \\ a_i & \text{for } x_i \in S_3, \end{cases} \tag{4.3}$$

and similarly

$$b_{i+1} = \begin{cases} b_i & \text{for } x_i \in S_1, \\ 2b_i \pmod n & \text{for } x_i \in S_2, \\ b_i + 1 \pmod n & \text{for } x_i \in S_3. \end{cases} \tag{4.4}$$

If the sets S_i are chosen more or less randomly, then the sequence x_i should behave more or less as a random walk through the group G. Moreover, if the sequence x_i were a random sequence of elements of G, then we should expect that there exists an integer $i \leq 3n^{1/2}$ such that $x_i = x_{2i}$ (see [55]). Such an integer may be found by computing the sequences $(x_i, a_i, b_i, x_{2i}, a_{2i}, b_{2i})$ recursively using the rules (4.2), (4.3), and (4.4). If we find that $x_i = x_{2i}$, then this gives us the desired equation (4.1) with $s = a_i - a_{2i} \pmod n$ and $t = b_{2i} - b_i \pmod n$.

If we are lucky and $\gcd(s, n) = 1$, then we simply compute an integer u such that $us \equiv 1 \pmod n$, and then we have $a = g^{ut}$, so that $\operatorname{ind}_g a = ut \pmod n$. If we are unlucky and find that $\gcd(s, n) = d > 1$, then we can still recover the answer without too much more work. We use the extended Euclidean algorithm to construct integers u and v such that $d = us + vn$. It follows from (4.1) that $a^d = g^{ut}$. Since the left side is a dth power, it follows that ut is divisible by d, giving an equation of the form $a^d = g^{dk}$. From this we obtain $a = g^{k+(in/d)}$ for some integer i with $1 \leq i \leq d$. We can now compute the discrete logarithm by simply checking the equation for $i = 1, 2, \ldots$ until the correct value is found. Note that if s were a random residue modulo n, then we would expect large values of d to be rare, so that the dominant factor of the running time usually comes from the time to find i.

It should be noted that up until very recently (see §7) the preceding algorithms were the only ones known for the case where G is the group of points on an elliptic curve over a finite field. For this reason, these groups have been suggested as good candidates for use in discrete logarithm cryptosystems (see [34], [36], and [49]). It has also been suggested to use Jacobians of higher dimensional abelian varieties [35].

4.2. Algorithms for groups with smooth orders. A positive integer is called *smooth* if it has no large prime factors, or more specifically, it is called *y-smooth* if it has no prime factors exceeding y. In this section we shall describe an algorithm that works well in the case that the order of g is smooth. The algorithm described here was discovered by S. Pohlig and M. Hellman [54], and independently by Roland Silver.

Let g have order n as before, and let

$$n = \prod_{i=1}^{k} p_i^{c_i}, \quad p_1 < \cdots < p_k. \tag{4.5}$$

Because $\operatorname{ind}_g a$ is only determined modulo n, we can then compute it by the Chinese remainder theorem if we can compute it modulo $p_i^{c_i}$. For simplicity of notation, let p be a prime with $p^c \mid n$ and $p^{c+1} \nmid n$, and let $x = \operatorname{ind}_g a \pmod{p^c}$. In order to determine x, it suffices to calculate b_j with

$$x = \sum_{j=0}^{c-1} b_j p^j, \quad 0 \le b_j < p.$$

Note that $\operatorname{ind}_g a = x + p^c t$ for some integer t, so that

$$\begin{aligned} a^{n/p} &= g^{nx/p + np^{c-1}t} \\ &= g^{n\sum_{j=0}^{c-1} b_j p^{j-1}} \\ &= g^{nb_0/p}. \end{aligned}$$

In order to compute b_0, we first calculate $a^{n/p}$ and $\zeta = g^{n/p}$. We then compute ζ^i for $i = 0, \ldots$ until we find $b_0 = i$ with $\zeta^i = a^{n/p}$.

If $c > 1$, then we can compute b_1 as follows. We first compute $h = g^{-1} = g^{n-1}$. We now set $a_1 = ah^{b_0}$ and note that

$$\begin{aligned} a_1^{n/p^2} &= g^{n\sum_{j=1}^{c-1} b_j p^{j-2}} \\ &= \zeta^{b_1}. \end{aligned}$$

We now simply search through the powers of ζ until the proper value of b_1 is found. Continuing in this manner we can calculate $b_0, \ldots, b_c$.

We now estimate the number of group operations required for the algorithm. For n given by (4.5), the number of group operations to calculate $\operatorname{ind}_g a \pmod{p_i^{c_i}}$ can easily be shown to be

$$O\left(\sum_{i=1}^{k} c_i(\log n + p_i)\right),$$

provided we use the fast exponentiation method described in [**32**, §4.6.3]. Once this is done, the Chinese remainder theorem can be applied to derive the final value of $\operatorname{ind}_g a$. (For a complexity analysis of the Chinese remainder theorem, see [**3**, §8.6 and 8.11].) It should be noted that there is a way to reduce the number of group operations by increasing the storage space, using the method of Shanks described in §4.1 to speed up the search for the correct powers of ζ. With this change, the running time is reduced to

$$O\left(\sum_{i=1}^{k} c_i(\log n + p_i^{1/2} \log p_i)\right)$$

group operations, but the storage space increases to $O(p_k^{1/2})$ group elements. The details for this appear in [**54**].

The Silver-Pohlig-Hellman algorithm shows that we should avoid groups for which the order has all small prime factors. For example, if the group is $\mathrm{GF}(2^k)^*$, then we should check the factorization of $2^k - 1$ to see that it has at least one large prime factor. Partial factorizations of $2^k - 1$ have been computed for $k \leq 1200$ [**12**], which should cover most of the values that are interesting for practical applications in the near future.

EXAMPLE. Consider using the multiplicative group of the finite field $\mathrm{GF}(2^{324})$, which has order $2^{324} - 1$. This is an extremely bad choice, since here the order of the group factors as

$$\begin{aligned} 2^{324} - 1 = {} & 3^5 \cdot 5 \cdot 7 \cdot 13 \cdot 19 \cdot 37 \cdot 73 \cdot 109 \cdot 163 \\ & \cdot 2593 \cdot 71119 \cdot 87211 \cdot 135433 \cdot 246241 \cdot 262657 \\ & \cdot 279073 \cdot 3618757 \cdot 97685839 \cdot 106979941 \cdot 168410989 \\ & \cdot 272010961 \cdot 4977454861. \end{aligned}$$

Since the largest prime factor in the order has only ten decimal digits, it is quite feasible to compute discrete logarithms using the Silver-Pohlig-Hellman algorithm.

If one is selecting a group of the form $\mathrm{GF}(p)^*$ for a prime p, then we need to choose a prime p for which $p-1$ has at least one large prime factor. This can be easily carried out by simply choosing a large prime q and selecting p as the smallest prime in the arithmetic progression $1 \pmod q$. A heuristic argument of Wagstaff [**62**] suggests that we will probably find such a prime before we examine the first $\log^2 q$ numbers in the arithmetic progression. For more information on how to efficiently determine if a large integer is prime, see the accompanying paper by A. K. Lenstra. It should be noted that for a prime constructed in this way, it is easy to find a primitive root modulo p (an example is given in §6).

5. The Index Calculus Method

In this section we discuss a class of algorithms that work very well when the group is endowed with a special structure. The approach is commonly known as the index calculus method, and apparently first appeared in the work of Kraitchik [**38**, pp. 119–123], [**39**, pp. 69–70, 216–267] and Cunningham (see [**63**]). The method was later rediscovered and analyzed by Adleman [**1**], Merkle [**48**], and Pollard [**55**]. In contrast to previous techniques, the index calculus method is probabilistic rather than deterministic.

In order to give a general description of the technique, let $p_1, \ldots, p_m$ be elements of G, and let G be generated by g and of order n. The algorithm has three stages. In the first stage of the algorithm, we gather identities of

the form

$$\prod_{j=1}^{m} p_j^{a_{ij}} = g^{b_i}. \tag{5.1}$$

This set of identities can be interpreted as a set of linear congruences

$$\sum_{j=1}^{m} a_{ij} \operatorname{ind}_g p_j \equiv b_i \pmod{n}. \tag{5.2}$$

In the second phase we solve the system for the $\operatorname{ind}_g p_j$. After these two initial stages, we compute individual logarithms in the third stage. In order to compute $\operatorname{ind}_g a$, we construct a relation of the form

$$\prod_{j=1}^{m} p_j^{e_j} = ag^e, \tag{5.3}$$

from which we obtain $\operatorname{ind}_g a = \sum_{j=1}^{m} e_j \operatorname{ind}_g p_j - e$. Note that the first two stages comprise a precomputation stage, after which we need only carry out the third stage for the calculation of individual logarithms. In practice this third stage takes considerably less time than the first two stages.

The reason that the index calculus method is not a general method is that it is not obvious how to generate the relations (5.1) for a general group in an efficient manner. At present such a method is only known for some finite fields and class groups of imaginary quadratic number fields. In the sections that follow, we shall describe the basic methods when applied to finite fields $\mathrm{GF}(p)$ for a prime p and $\mathrm{GF}(2^k)$. For the description of the algorithm in class groups, see **[45]**, and for a discussion of the algorithm as it applies to other classes of finite fields, see **[29]** and **[23]**.

5.1. Finite fields $\mathrm{GF}(p)$. Curiously enough, the reason that the index calculus method works well in $\mathrm{GF}(p)^*$ is that the problem of factoring is easy for many integers. In particular, if all of the prime factors of an integer k are less than a given bound t, then we can completely factor k with at most $t + \log k$ divisions.

In the index calculus method for $\mathrm{GF}(p)^*$, we take $p_1, \ldots, p_m$ to be the first m primes. We begin by describing the generation of relations in stage one. For this, we first choose a random integer $b \in [1, n]$ and compute the least positive integer r with $r = g^b \pmod{p}$. We then try to factor r as a product of the first m primes, using, for example, simple division by these primes. If r factors in this way, then we obtain a relation of the form (5.1).

It is evident that the bigger we choose m, the greater the chance that r will factor as a product of the first m primes. On the other hand, as m increases, the work to solve the system of equations in the second stage increases, and the work done to factor residues in the first stage increases. In order to optimize the running time of the algorithm, we need to balance

these constraints. We now sketch an argument concerning the running time of the algorithm.

If b is chosen from a uniform distribution, then the probability that r will factor as a product of the first m primes is $\psi(p, p_m)/p$, where $\psi(x, y)$ is defined to be the number of positive integers $\leq x$ that have no prime factors exceeding y. The asymptotic behavior of the function ψ has been extensively studied, and in particular it is known that (see [**16**])

$$\psi(x, y) = x \exp((-1 + o(1))u \log u),$$

where $u = \log x/\log y$, for $u \to \infty$ and $y \geq \log^2 x$. If we choose $p_m \approx L(p)^c$, where c is a constant and

$$L(p) = \exp(\sqrt{\log p \log\log p}),$$

then the probability that r has all prime factors among the first m primes is $L(p)^{-1/(2c)+o(1)}$. Hence we should expect to generate a relation of the form (5.1) after we try $L(p)^{1/(2c)+o(1)}$ values of b, and generating $2m$ such relations should take about

$$2mL(p)^{1/(2c)+o(1)} = L(p)^{c+1/(2c)+o(1)}$$

values of b. If we use trial division to do the factoring, then for each b it will take at most $m + \log p$ divisions to decide if it gives us a relation, so that we have a total running time for the generation of relations in the first stage of $L(p)^{2c+1/(2c)+o(1)}$.

Once we have generated $2m$ relations (or any amount slightly larger than m), it is reasonable to expect that the corresponding system of $2m$ equations in m unknowns should have full rank, so that we can solve for the $\operatorname{ind}_g p_j$. (Here when we say full rank, we mean full column rank modulo q for every prime q dividing $p - 1$.)

The problem of solving a system of linear congruences presents no serious difficulties, but there are a few points that deserve comment. Consider the problem of solving the system

$$\begin{aligned} 3x_1 + 7x_2 &\equiv 0 \pmod{42}, \\ 2x_1 - 3x_2 &\equiv 2 \pmod{42}. \end{aligned} \tag{5.4}$$

Note that none of the coefficients are invertible modulo 42, and it is impossible to add a multiple of one row to another in such a way as to introduce a zero entry (as we might if we tried Gaussian elimination). On the other hand, the system has the unique solution $x_1 \equiv 28 \pmod{42}$, $x_2 \equiv 18 \pmod{42}$.

Let us now describe a method for solving systems of linear congruences. Suppose we want to solve $Ax \equiv b \pmod{p-1}$, where A is $k \times m$ with $\operatorname{rank}(A \bmod q) = m$ for every prime q dividing $p - 1$. One technique is as follows. First factor $p - 1$ as a product of prime powers. Then for each prime q dividing $p - 1$ we solve the system modulo q using Gaussian elimination (note that $\mathbb{Z}/q\mathbb{Z}$ is a field). We then use Hensel's method to lift

the solution modulo q to a solution modulo the power of q dividing $p-1$. Finally we use the Chinese remainder theorem to combine the solutions for a solution modulo $p-1$. It turns out, however, that the factorization of $p-1$ is not needed to solve the system and should probably be avoided. One alternative is to proceed as if $\mathbb{Z}/(p-1)\mathbb{Z}$ is a field and attempt to perform Gaussian elimination in the usual manner. This can, however, break down if we find a column in which no entry is invertible modulo $p-1$ (as in (5.4)). It is possible to describe a method to recover from this using the Chinese remainder theorem, the Euclidean algorithm, and Hensel's method, but the details are somewhat tedious so we shall adopt another strategy.

In standard Gaussian elimination, we would search the j th column of the matrix to find an entry that is invertible modulo $p-1$, exchange rows to bring it into the jj location, and add a multiple of the j th row to the rows below it to introduce zero entries. An alternative procedure is as follows: search the j th column to find an entry that is nonzero, and exchange rows to bring it into the jj th location. Then, in order to introduce a zero into the ij th location, we first use the extended Euclidean algorithm (see [32, §4.5.2]) to find integers g, r, and s for which $g = \gcd(a_{ij}, a_{jj}) = ra_{ij} + sa_{jj}$. We then replace row j of the matrix A by $r\cdot(\text{row } i)+s\cdot(\text{row } j)$, and replace row i by $(a_{ij}/g)\cdot(\text{row } j)-(a_{jj}/g)\cdot(\text{row } i)$. This operation on the system preserves the solution set and has the effect of replacing the jj th entry by g and replacing the ij th entry by 0.

If we use this approach, the solution of the system of equations will take $O(m^3)$ operations modulo $p-1$, and $O(m^2)$ applications of the extended Euclidean algorithm. Hence the total expected time for stages one and two to solve for the $\operatorname{ind}_g p_j$'s is

$$L(p)^{2c+1/(2c)+o(1)} + L(p)^{3c+o(1)}$$

operations on integers of size p. Since each such operation can be done in $L(p)^{o(1)}$ bit operations, we get the same number of bit operations. By choosing $c = 1/2$ we get a running time for the first two stages of $L(p)^{2+o(1)}$ bit operations.

It is in the third stage where we compute individual discrete logarithms. In order to compute $\operatorname{ind}_g a$, we choose a random integer e, compute $r = ag^e \pmod p$, and see if r factors as a product of the first m primes. If we obtain $r = \prod_{j=1}^{m} p_j^{e_j}$, then this implies

$$\operatorname{ind}_g a \equiv \sum_{j=1}^{m} e_j \operatorname{ind}_g p_j - e \pmod{p-1}.$$

The third stage can be analyzed in the same manner as before, giving an expected running time of $L(p)^{c+1/(2c)+o(1)}$, or $L(p)^{3/2+o(1)}$.

5.1.1. Rigorously analyzed versions. While the discussion in the preceding section gives a flavor for the subject of the index calculus method in $\mathrm{GF}(p)^*$,

there were several issues left unresolved. The first of these is the fact that we ignored the issue of why the set of equations should have full rank. If the procedure is modified slightly then this can be rigorously proved; for the details see the paper of Pomerance [**56**]. There are also several ways in which the running time can be improved. One of these is to use a faster method for doing the factorizations. If we use a rigorously analyzed version of H. W. Lenstra's elliptic curve method, then the running time for the first stage can be reduced to $L(p)^{\sqrt{2}+o(1)}$. If in addition we use sparse matrix techniques such as those of [**64**] to solve the system of linear equations, then we also reduce the running time of the second stage to $L(p)^{\sqrt{2}+o(1)}$. After this precomputation, the computation of individual logarithms in the third stage can be carried out in $L(p)^{1/\sqrt{2}+o(1)}$ bit operations. These improvements are due to Pomerance [**56**]. It should also be noted that if the running time of the first two stages is increased and we use the elliptic curve factorization method in the third phase, then the time for the last stage can be decreased further.

5.1.2. Trading rigor for speed. The algorithm of Pomerance has, to date, the smallest asymptotic running time that is rigorously proved for $\mathrm{GF}(p)^*$, but there are variations of the index calculus method due to Coppersmith, Odlyzko, and Schroeppel [**20**], which are conjectured to be faster. In their paper they actually describe three such algorithms, called the linear sieve, the residue list sieve, and the Gaussian integer method. For each of these methods there is a heuristic analysis that suggests a running time of $L(p)^{1+o(1)}$ for the first two stages, followed by a running time of $L(p)^{1/2+o(1)}$ to compute individual logarithms in the third stage. Unfortunately, none of them have been rigorously analyzed, so their running times remain conjectures. In this treatment we shall be content to describe the Gaussian integer method and give a heuristic analysis of the running time.

For simplicity consider the case where $p \equiv 1 (\bmod 4)$, so that -1 is a quadratic residue modulo p. The description of the algorithm takes place in an isomorphic copy of $\mathrm{GF}(p)^*$. We construct such a copy by first finding integers T and V with $T^2 + V^2 = p$. This can be done by a method related to the Euclidean algorithm (see [**13**]); note that trivially we have $0 \le T, V \le \sqrt{p}$. Now consider the ring $\mathbb{Z}[i]/(T + iV)$. Since $(T + iV)$ is a maximal ideal, the ring $\mathbb{Z}[i]/(T + iV)$ is actually a field, so it is an isomorphic copy of $\mathrm{GF}(p)^*$. In fact, an explicit isomorphism is given by

$$\phi : \mathbb{Z}[i]/(T + iV) \longrightarrow \mathbb{Z}/p\mathbb{Z}, \quad \phi(c + id) = c + TV^{-1}d (\bmod p).$$

Let $b = e + if$ be a primitive element in the field. We take $p_1, \ldots, p_m$ to be a set of primes in $\mathbb{Z}[i]$ of norm $\le L(p)^{1/2}$ together with V, and we calculate the logarithms of the $p_i's$ to the base b. Using the isomorphism ϕ gives a method for calculating logarithms to the base $\phi(b) = e + TV^{-1} f (\bmod p)$.

In the first stage we shall construct relations by a sieving procedure, searching for integers $c, d \leq L(p)^{1/2}$ such that $cV - dT$ is smooth with respect to $L(p)^{1/2}$. The sieving is accomplished by fixing a value of c and sieving the values of d. The sieving starts by initializing to all zeros an array of length $L(p)^{1/2}$. For each (real) prime power p_i^j such that $p_i \nmid T$ and $p_i^j \leq L(p)^{1/2}$, find the least positive integer d with $d \equiv cVT^{-1} (\mathrm{mod}\, p_i^j)$. We then increase the array value corresponding to d by an approximation to $\log p_i$, and then stepping through the array in steps of p_i^j, we perform the same operation on all array elements corresponding to the integers $\equiv d (\mathrm{mod}\, p_i^j)$. In addition, for the largest possible value of j, we check each of the integers that are sieved to see if they are divisible by higher powers of p_i, and if so we add the appropriate multiple of $\log p_i$.

After this process, the array elements contain approximations to the logarithms of the "smooth part" of the integers $cV - dT$. We then search through the array to find those entries that are large enough to indicate that they probably correspond to smooth residues. From these we can then use trial division to check if they do in fact correspond to smooth residues, and if so then we obtain a relation of the form

$$\prod_{\substack{j=1 \\ p_j \text{ real}}}^{m} p_j^{a_{ij}} \equiv cV - dT (\mathrm{mod}\, p).$$

For each such pair (c, d), it is easy to see that

$$cV - dT \equiv V(c + id) (\mathrm{mod}\, T + iV),$$

and if it happens that $c + id$ factors as a product of complex primes of norm not exceeding $L(p)^{1/2}$, then we obtain a linear relation modulo $p - 1$ involving the discrete logarithms of these primes (and V) to the base b.

Note that the complex number $c + id$ is quite small since $c, d \leq L(p)^{1/2}$. Hence we might expect that the probability of $c + id$ factoring completely using the complex primes of norm $\leq L(p)^{1/2}$ is bounded below by a constant (see [**28**]). The numbers $cV - dT$ are only slightly larger than $\sqrt{p}$ in magnitude, and therefore have a probability of $L(p)^{-1/2+o(1)}$ that they will be smooth. Hence we probably need only examine $L(p)^{1+o(1)}$ pairs c, d to get $L(p)^{1/2}$ equations, and the running time of stage 1 is $L(p)^{1+o(1)}$. Using sparse matrix techniques, the running time for stage 2 will also be $L(p)^{1+o(1)}$. Stage 3 can be performed in much the same way as stage 1, giving a running time of $L(p)^{1/2+o(1)}$. Unfortunately, this analysis is only heuristic, since we have no argument to show that the events of $c + id$ and $cV - dT$ both being smooth are independent.

While it is perhaps premature to mention it, the Gaussian integer algorithm has recently spurred some very interesting developments on other discrete logarithm and factoring algorithms. First of all, it inspired J. Pollard

to develop a method for factoring integers that are very close to cubes of integers. The method of Pollard requires computations to be performed in a cubic extension of the rationals, and has therefore come to be known as the number field sieve (see [**42**]). This in turn led several people to realize that the method could be generalized to work for arbitrary integers, using arithmetic in more general extensions of the rationals. At the time of this writing, much work is left to be done to determine if the general method can ever be made practical or rigorous. Investigations by Joe Buhler, H. W. Lenstra, Jr., and Carl Pomerance have at least uncovered a heuristic argument to show that the method should be able to factor an integer n in approximately $\exp\left(c(\log n)^{1/3}(\log\log n)^{2/3}\right)$ operations for some constant c. Further work by Dan Gordon [**26**] suggests that the technique can also be applied to the problem of computing discrete logarithms in $\mathrm{GF}(p)^*$, resulting in an algorithm with heuristic running time $\exp\left(c(\log p)^{1/3}(\log\log p)^{2/3}\right)$.

5.1.3. Practical experience in $\mathrm{GF}(p)^*$. The emphasis in this section has been on asymptotic analysis of running times. Just as in integer factoring algorithms, there are numerous refinements that can be applied to improve the running times of practical versions. In fact there has been rather little practical experience with such algorithms, and the only serious attempt to implement an index calculus method for $\mathrm{GF}(p)^*$ was carried out very recently by Brian LaMacchia and A. M. Odlyzko [**40**] of Bell Laboratories, who have implemented the Gaussian integer method of [**20**] and the Lanczos method for sparse linear systems over a finite field. Using their programs they were able to build a dictionary of approximately 90,000 logarithms for a prime of 58 digits (specially chosen as one used in an actual system). Their experience seems to suggest that it is possible to compute discrete logarithms in groups $\mathrm{GF}(p)^*$ with $p \approx 10^{100}$.

5.2. Finite fields $\mathrm{GF}(2^k)$. In discussing discrete logarithms over $\mathrm{GF}(2^k)^*$, we should first agree on a model for the group. It is convenient to regard $\mathrm{GF}(2^k)$ as consisting of the set of polynomials over $\mathrm{GF}(2)$ of degree less than k, with operations performed modulo some fixed irreducible polynomial $f(x)$ of degree k. In the case of $\mathrm{GF}(p)^*$, we took the group elements $p_1, \ldots, p_m$ to be the first m small primes. In the case of $\mathrm{GF}(2^k)^*$, we take $p_1, \ldots, p_m$ to be the irreducible polynomials of degree not greater than some value t. The relationship between t and m is easily deduced from the fact that the number of irreducible polynomials over $\mathrm{GF}(2)$ of degree less than or equal to t is exactly [**32**, ex. 4.6.2.4]

$$\sum_{d \le t} d^{-1} \sum_{f \mid d} \mu(f) 2^{d/f},$$

which is approximately $2^{t+1}/t$.

Let g be a generator of $\mathrm{GF}(2^k)^*$. In the first stage of the algorithm,

we choose random integers $b \in [1, 2^k - 1]$, compute a polynomial r with degree $(r) < k$ and $r = g^b$, and try to factor r as a product of irreducibles of degree not exceeding t. This can be done relatively quickly using a fast exponentiation method and a fast method for factoring polynomials. It is perhaps surprising to learn that the problem of factoring polynomials over finite fields is relatively easy, and that there exist probabilistic algorithms that will factor a polynomial over GF(2) of degree k in an expected time of $O(k^c)$ for some constant c (see [**32**, §4.6.2]).

In order to estimate the running time of the first stage, we need to know how many polynomials over GF(2) have all of their irreducible factors of degree at most t. We let $N(d, t)$ denote the number of polynomials of degree exactly d, all of whose irreducible factors are of degree not exceeding t. Since the polynomials r that are generated are random polynomials of degree $< k$, the probability that a given value of b will produce a relation of the form (5.1) is

$$\frac{\sum_{d<k} N(d, t)}{\sum_{d<k} N(d, d)}.$$

Odlyzko [**50**] has shown that this latter quantity is of the form

$$\exp\left((1 + o(1))\frac{k}{t}\log_e\left(\frac{t}{k}\right)\right),$$

provided $k \to \infty$ and $k^{1/100} < t < k^{99/100}$. Hence, in order to generate m relations of the form (5.1), we expect stage 1 to require examination of

$$m \exp\left((1 + o(1))\frac{k}{t}\log_e\left(\frac{k}{t}\right)\right)$$

values of b. Since each b takes time $O(k^c)$ operations for a constant c, the total running time for stage 1 is of the same form.

The second stage of the algorithm takes $m^{2+o(1)}$ operations modulo $2^k - 1$ if we use the sparse matrix methods of Wiedemann [**64**], giving a total running time for the first two stages of

$$m \exp\left((1 + o(1))\frac{k}{t}\log_e\left(\frac{k}{t}\right)\right) + O(m^{2+o(1)}k^c).$$

By choosing

$$t \approx \left(\frac{k \log_e k}{2 \log_e 2}\right)^{1/2},$$

we get a total running time of $\exp\left((c_1 + o(1))(k \log_e k)^{1/2}\right)$, where $c_1 = \sqrt{2 \log_e 2}$. The time for the third stage can be analyzed in a similar manner, giving a running time of $\exp\left((\frac{1}{2c_1} + o(1))(k \log_e k)^{1/2}\right)$. This is in fact the best rigorously proved running time for an algorithm in $\mathrm{GF}(2^k)^*$ (see [**56**]).

So far, the description of the index calculus method as it applies to $GF(2^k)^*$ has paralleled exactly the presentation for $GF(p)^*$. In the case of $GF(2^k)^*$ there are, however, some very significant improvements that can be made, both in the performance of implementations and the heuristic asymptotic running time estimates. Blake, Fuji-Hara, Mullin, and Vanstone [7] gave several such improvements, including one based on the fact that for any integer b, it is relatively easy to calculate polynomials h_1 and h_2 of degree $\leq k/2$ with $h_1 g^b = h_2$. If all of the irreducible factors of both h_1 and h_2 have degree less than t, then we obtain a relation of the form (5.1). Since their degree is smaller, this is slightly more likely to happen, even though we require them both to have this property. This improvement does not significantly improve the asymptotic behavior, however.

The improvement of Blake, et al, relies on the factorization of polynomials of degree less than $k/2$, and there are several other methods that take this approach. A complete description of these developments is beyond the scope of this paper, but the interested reader will find an excellent description in the survey paper by Odlyzko **[50]**. In the present paper we shall be content to describe a significant improvement on the basic index calculus method due to Coppersmith **[18]**. Coppersmith's algorithm relies on factorizations of polynomials of degree approximately $k^{2/3}$, and is interesting for both practical and theoretical reasons. The present arguments for the asymptotic running time are based on heuristic assumptions, and it remains an open question to prove that the analysis is correct. The heuristic argument suggests that it will have a running time of $\exp\left(ck^{1/3}\log^{2/3} k\right)$ for some constant c. In practice, the algorithm performs much better than previous algorithms (see **[19]**, **[18]**), and there seems to be little reason to doubt the heuristic arguments.

In the Coppersmith algorithm, we assume that the polynomial f used to define the field has the form $x^k + f_1(x)$, where the degree of f_1 is of size $\log k$. It turns out that this is not a severe restriction, since it is relatively easy to transfer a discrete logarithm from one representation of the field to any other (see **[50]**, §5.2]). Moreover, there is a heuristic argument to suggest that such irreducible polynomials should exist (although this remains unproved).

In order to describe the first stage in the Coppersmith method, we shall require some notation. Let r be an integer, and define $h = \lfloor k2^{-r} \rfloor + 1$. To generate a relation, we first choose random relatively prime polynomials $A(x)$ and $B(x)$ of degrees $\leq t$. We then set $w_1(x) = A(x)x^h + B(x)$ and

$$w_2(x) = w_1(x)^{2^r} \pmod{f(x)}. \tag{5.5}$$

It follows from our special choice of $f(x)$ that we can take

$$w_2(x) = A(x^{2^r})x^{h2^r - k} f_1(x) + B(x^{2^r}),$$

so that $\deg(w_2) \leq 2^r t + h2^r - k + \deg(f_1)$. If we choose t and 2^r to be of

order $k^{1/3}$, then the degrees of w_1 and w_2 will be of order $k^{2/3}$. If they behave as random polynomials of that degree (as we might expect), then there is a good chance that all their irreducible factors will have degrees not exceeding t. If so, then from (5.5) we obtain a linear equation involving the logarithms of polynomials of degree $\leq t$.

A detailed analysis of the method shows that the parameters can be chosen to give a running time for the first stage of

$$\exp((c_2 + o(1))k^{1/3}\log^{2/3} k), \qquad \text{where } c_2 < 1.351.$$

The second stage of the Coppersmith method is the same as in the basic index calculus method, requiring a solution of a system of linear congruences. The third stage is somewhat more complicated than in the basic index calculus method. We omit the details of how it works, but the basic idea is to compute the logarithm of an individual polynomial by computing the logarithms of a sequence of polynomials with decreasing degrees. The running time of the third phase is of the form

$$\exp((c_3 + o(1))k^{1/3}\log^{2/3} k),$$

where $c_3 < 1.098$, so it takes less time than the first two stages.

The previous discussion was primarily concerned with asymptotic analysis of algorithms, but it is interesting to note that both Blake, et al [7] and Coppersmith and Davenport [**19**] have implemented their algorithms for the test case $\mathrm{GF}(2^{127})^*$, and have been successful in building a database of logarithms that will enable them to rather easily compute individual logarithms. This is significant since both Mitre and Hewlett-Packard have in the past chosen this field for systems intended for actual use, and they must now be regarded as totally insecure. Odlyzko [**50**] has carried out an extensive analysis that suggests the Coppersmith algorithm will make it feasible to compute discrete logarithms in $\mathrm{GF}(2^k)^*$ for $k < 520$ with a supercomputer, and perhaps for $k < 1280$ using very expensive special purpose hardware. It is interesting to note that a Canadian company named Newbridge Microsystems is now producing a Data Encryption Processor chip that implements arithmetic in $\mathrm{GF}(2^{593})$. This chip is intended for use in various cryptographic protocols whose security is based on the difficulty of the discrete logarithm problem in $\mathrm{GF}(2^{593})$.

6. An Example and a Challenge

Let $q = (7^{149} - 1)/6$, and let $p = 2 \cdot 739 \cdot q + 1$. A summary of a proof that q is prime may be found in [**12**]. Moreover, it is easy to verify that

$$7^{(p-1)/2} \not\equiv 1 \pmod{p},$$
$$7^{(p-1)/739} \not\equiv 1 \pmod{p},$$
$$7^{(p-1)/q} \not\equiv 1 \pmod{p}.$$

It follows from these equations that 7 has order $p-1$ modulo p, so that 7 must be a primitive root modulo p, and from this it easily follows that p is prime.

As an example of how this can be used in a Diffie-Hellman scheme, party A chooses a random residue x modulo p, computes $7^x \pmod{p}$, and sends the result to party B, keeping x secret. B receives

$$7^x = 12740218011997394682426924433432284974938204258693162165455773529032291467909599868186097881304659516645545814428058807676603378 1.$$

Party B then chooses a random residue y modulo p, computes $7^y \pmod{p}$, and sends the result to A, keeping y secret. A receives

$$7^y = 18016228528745310244478283483679989501596704669534669731302512173405995377205847595817691062538069210165184866236213793402680304 9.$$

They can now both compute the secret key $7^{xy} \pmod{p}$.

Now for a challenge: The author will pay \$100 to the first person who finds the secret key constructed from the above communication. This offer will only be paid if the person claiming it provides a proof that their answer is correct!

7. Some Open Questions

There remain a number of interesting open questions regarding discrete logarithms. The biggest one of these is of course whether the discrete logarithm problem is really "hard." At present the only evidence for this is our ignorance. We know that the problem has been studied going back at least as far as Gauss [**25**] in 1801 and Jacobi [**31**] in 1839, but the algorithms that are presently known are not fast enough to break the systems. Leaving this basic question aside for a moment, we see several other open questions. To this author, the most interesting of these are the following.

- Is there a sense in which the general discrete logarithm problem is equivalent to the Diffie-Hellman problem? (i.e., is it possible to generalize the result of [**9**]?).
- Are there other cases for which the discrete logarithm problem is easy, similar to the case when the group has smooth order? For example, is there a rapid way to calculate discrete logarithms in $\mathrm{GF}(p)^*$ when $p+1$ has all small prime factors?
- Very recently, Menezes, Vanstone, and Okamoto [**47**] have shown that the discrete logarithm problem for an elliptic curve group over a finite field $\mathrm{GF}(q)$ can be reduced to the discrete logarithm problem in $\mathrm{GF}(q^k)^*$ for some k. For supersingular curves, it turns out that k is rather small, and for this case we obtain an algorithm with subexponential running time. One question remains: Is there an

algorithm for discrete logarithms in elliptic curve groups to improve on the Shanks/Pollard running time estimates?

- For groups $GF(p^k)^*$ with fixed p, the methods for $GF(2^k)^*$ carry over directly to give an algorithm with "subexponential" running time (see [29]). The case $GF(p^2)^*$ was also considered by ElGamal [23]. For other finite fields there is a gap in our knowledge, and in particular, for the groups $GF(p^k)^*$ where both p and k tend to infinity, it is unknown whether there is a subexponential algorithm.
- Can one find a variation of Coppersmith's algorithm for $GF(2^k)^*$ that has a rigorously proved running time of $\exp(ck^{1/3}\log^{2/3}k)$ for some constant c?
- Is there an algorithm for computing discrete logarithms in $GF(p)^*$ with a rigorously provable running time of $L(p)^{1+o(1)}$? There are several algorithms that have been conjectured to have this running time in [20], but none of them have ever been fully analyzed. An even more ambitious project would be to prove that the method described in [26] based on the number field sieve has the conjectured expected running time of $\exp(c(\log p)^{1/3}(\log\log p)^{2/3})$ for some constant c.
- Can one prove that a positive proportion of the bits of the discrete logarithm in $GF(p)^*$ are simultaneously secure in the sense of [43]? (See §3.)

The discrete logarithm problem has many different facets to it, and there are areas in which much remains to be discovered. If the reader is interested in learning more about the subject of discrete logarithms, he/she may profit by consulting some of the references given here. The list of references is not a complete bibliography on the subject, but contains a good sample of papers on the three themes covered in this survey, namely applications to cryptography, algorithms for computing discrete logarithms, and other complexity issues. Of particular interest is the survey paper of Odlyzko [50], in which he gives a nice treatment of algorithms for the discrete logarithm problem with emphasis on $GF(2^k)$, along with numerous other useful references. Furthermore, anyone interested in modern cryptography should definitely read the original paper of Diffie and Hellman [21] that opened Pandora's box on applications of number theory to cryptology. This is probably the paper that is now cited most often in research papers on cryptology.

References

1. L. M. Adleman, *A subexponential algorithm for the discrete logarithm problem with applications to cryptography*, Proc. of the 20th Annual IEEE Symposium on Foundations of Computer Science (1979), 55–60.
2. L. M. Adleman and K. S. McCurley, *Open problems in number theoretic complexity*, Discrete Algorithms and Complexity; Proc. of the Japan–U.S. Joint Seminar, Academic Press, Orlando, Florida, 1987, pp. 237–262.
3. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The design and analysis of*

computer algorithms, Addison-Wesley, Reading, MA, 1974.
4. Eric Bach, *Discrete logarithms and factoring*, Technical Report UCB/CSD 84/186, Computer Science Division (EECS), University of California, Berkeley, California, June, 1984.
5. ——, *Intractable problems in number theory*, Adv. in Cryptology: Proc. of Crypto '88, Lecture Notes in Computer Science, vol. 403, Springer-Verlag, NY., 1990, pp. 73–93.
6. Thomas Beth, *Efficient zero-knowledge identification scheme for smart cards*, Adv. in Cryptology (Proc. of Eurocrypt '88), Lecture Notes in Computer Science, vol. 330, Springer-Verlag, New York, 1988, pp. 77–84.
7. I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone, *Computing logarithms in fields of characteristic two*, SIAM J. Algebraic Discrete Methods **5** (1984), 276–285.
8. M. Blum and S. Micali, *How to generate cryptographically strong sequences of pseudo-random bits*, SIAM J. of Comput. **13** (1984), 850–864.
9. Bert den Boer, *Diffie-Hellman is as strong as discrete log for certain primes*, Proc. of Crypto '88, Lecture Notes in Computer Science, vol. 403, Springer-Verlag, New York, 1990, pp. 530–539.
10. Gilles Brassard, *A note on the complexity of cryptography*, IEEE Trans. Inform. Theory **25** (1979), 232–233.
11. E. F. Brickell, P. J. Lee, and Y. Yacobi, *Secure audio teleconference*, Proc. of Crypto '87, Lecture Notes in Computer Science, vol. 293, Springer-Verlag, NY, 1988, pp. 418–426.
12. John Brillhart, D. H. Lehmer, J. L. Selfridge, Bryant 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*, (second edition) Contemporary Mathematics, vol. 22, American Mathematical Society, Providence, 1988.
13. John Brillhart, *Note on representing a prime as a sum of two squares*, Math. Comp. **26** (1972), 1011–1013.
14. Johannes Buchmann and Hugh C. Williams *A secure key exchange system based on imaginary quadratic fields*, J. Cryptology **1** (1988), 107–118.
15. ——, *A key exchange system based on real quadratic fields*, Proc. of Crypto '89, Lecture Notes in Computer Science, vol. 435, Springer-Verlag, NY, 1990, pp. 335–343.
16. E. R. Canfield, P. Erdös, and C. Pomerance, *On a problem of Oppenheim concerning Factorisatio Numerorum*, J. Number Theory **17** (1983), 1–28.
17. David Chaum, Jan-Hendrik Evertse, and Jeroen van de Graf, *An improved protocol for demonstrating possession of discrete logarithms and some generalizations*, Adv. in Cryptology (Proceedings of Eurocrypt '87), Lecture Notes in Computer Science, vol. 304, Springer-Verlag, NY, pp. 127–142.
18. D. Coppersmith, *Fast evaluation of discrete logarithms in fields of characteristic two*, IEEE Trans. Inform. Theory **30** (1984), 587–594.
19. D. Coppersmith and J. H. Davenport, *An application of factoring*, J. Symbolic Comput. **1** (1985), 241–243.
20. D. Coppersmith, A. Odlyzko, and R. Schroeppel, *Discrete logarithms in* GF(p), Algorithmica **1** (1986), 1–15.
21. W. Diffie and M. Hellman, *New Directions in Cryptography*, IEEE Trans. Inform. Theory **22** (1976), 472–492.
22. T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Trans. Inform. Theory **31** (1985), 469–472.
23. ——, *A subexponential-time algorithm for computing discrete logarithms over* $GF(p^2)$, IEEE Trans. Inform. Theory **31** (1985), 473–481.
24. M. A. Frumkin, *Complexity questions in number theory*, J. Soviet Math. **29** (1985), 1502–1517.
25. K. F. Gauss, *Disquisitiones Arithmeticæ*, Leipzig, Fleischer, 1801. Translation into English by Arthur A. Clarke, S.J. reprinted by Springer-Verlag, New York, 1985.
26. Daniel M. Gordon, *Discrete logarithms in* $GF(p)$ *using the number field sieve*, preprint, 1990.
27. Cristoph G. Günther, *Diffie-Hellman and ElGamal protocols with one single authentica-*

tion key, to appear in Adv. in Cryptology (Proc. of Eurocrypt '89), Lecture Notes in Computer Science.
28. D. G. Hazlewood, *Gaussian integers with small prime factors*, Internat. J. Math. Math. Sci. **2** (1979), 81–90.
29. M. E. Hellman and J. M. Reyneri, *Fast computation of discrete logarithms in* $GF(q)$, Adv. in Cryptology: Proc. of Crypto '82, D. Chaum, R. Rivest, and A. Sherman, eds., Plenum Press, 1983, 3–13.
30. Shigeo Tsujii and Toshiya Itoh, *An ID-based cryptosystem based on the discrete logarithm problem*, IEEE J. on Selected Areas in Communications **8** (1989), 467–473.
31. C. G. J. Jacobi, *Canon arithmeticus*, Typis Academicis, Berlin, 1839.
32. D. E. Knuth, *The art of computer programming, Vol.* 2: *Seminumerical algorithms*, second edition, Addison-Wesley, Reading, MA, 1981.
33. ——, *The art of computer programming, Vol.* 3: *Sorting and searching*, Addison-Wesley, Reading, MA, 1973.
34. Neal Koblitz, *Elliptic curve cryptosystems*, Math. Comp. **48** (1987), 203–209.
35. ——, *A family of Jacobians suitable for discrete log cryptosystems*, to appear in Proc. of Crypto '88, Lecture Notes in Computer Science, vol. 403, Springer-Verlag, New York, 1990, pp. 94–99.
36. ——, *A course in number theory and cryptography*, Springer-Verlag, New York, 1987.
37. Kenji Koyama and Kazuo Ohta, *Identity-based conference key distribution systems*, Proc. of Crypto '87, Lecture Notes in Computer Science, vol. 293, Springer-Verlag, New York, 1988, pp. 175–184.
38. M. Kraitchik, *Théorie des nombres*, Vol. 1, Gauthier-Villars, Paris, 1922.
39. ——, *Recherches sur la théorie des nombres*, Gauthier-Villars, Paris, 1924.
40. B. A. LaMacchia and A. M. Odlyzko, *Computation of discrete logarithms in prime fields*, preprint, 1990.
41. A. K. Lenstra and H. W. Lenstra, Jr., *Algorithms in number theory*, Technical Report 87-008, Department of Computer Science, University of Chicago, May 1987.
42. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard, *The number field sieve*, Proc. of the 22nd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, 564–572.
43. Douglas L. Long and Avi Wigderson, *The discrete logarithm hides* $O(\log n)$ *bits*, SIAM J. Comput. **17** (1988), 363–372.
44. J. L. Massey, *Logarithms in finite cyclic groups—cryptographic issues*, Proc. of the 4th Annual Benelux Symposium on Information Theory (1983), 17–25.
45. Kevin S. McCurley, *Cryptographic key distribution and computation in class groups, Number theory and applications* (Proc. of the NATO Advanced Study Institute on Number Theory and Applications, Banff, 1988), Richard A. Mollin, ed., Kluwer, Boston, 1989, 459–479.
46. ——, *A key distribution system equivalent to factoring*, J. of Cryptology **1** (1988), 95–105.
47. Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto, *Reducing elliptic curve logarithms to logarithms in a finite field*, preprint, 1990.
48. R. Merkle, *Secrecy, authentication, and public key systems*, Ph.D. dissertation, Electrical Engineering Department, Stanford University, 1979.
49. V. Miller, *Use of elliptic curves in cryptography*, Adv. in Cryptology (Proc. of Crypto '85), Lecture Notes in Computer Science, vol. 218, Springer-Verlag, New York, 1986, pp. 417–426.
50. A. M. Odlyzko, *Discrete logarithms in finite fields and their cryptographic significance*, Adv. in Cryptology (Proc. of Eurocrypt 84), Lecture Notes in Computer Science, vol. 209, Springer-Verlag, New York, pp. 224–314.
51. R. W. K. Odoni, V. Varadharajan, and P. W. Sanders, *Public key distribution in matrix rings*, Electronics Letters **20** (1984), 386–387.
52. Eiji Okamoto, *Key distribution systems based on identification information*, Proc. of Crypto '87, Lecture Notes in Computer Science, vol. 293, Springer-Verlag, New York, 1988, pp. 194–202.
53. René Peralta, *Simultaneous security of bits in the discrete log*, Adv. in Cryptology (Proc.

of Eurocrypt '85), Lecture Notes in Computer Science, vol. 219, Springer-Verlag, New York, 1986, pp. 62–72.
54. Stephen C. Pohlig and Martin E. Hellman, *An improved algorithm for computing logarithms over* GF(p) *and its cryptographic significance*, IEEE Trans. on Inform. Theory **24** (1978), 106–110.
55. J. M. Pollard, *Monte Carlo methods for index computation mod* p, Math. Comp. **32** (1978), 918–924.
56. Carl Pomerance, *Fast, rigorous factorization and discrete logarithm algorithms*, in Discrete algorithms and complexity; Proc. of the Japan–U.S. Joint Seminar, June 4, 1986, Kyoto, Japan, Academic Press, Orlando, 1987, pp. 119–143.
57. H. Riesel, *Some soluble cases of the discrete logarithm problem*, BIT **28** (1989), 831–851.
58. C. P. Schnorr, *Efficient identification and signatures for smart cards*, Adv. in Cryptology (Proc. of Crypto '89), Lecture Notes in Computer Science, Springer-Verlag, NY, 1990, pp. 239–252.
59. R. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public key cryptosystems*, Comm. of the ACM **21** (1978), 120–126.
60. A. W. Schrift and A. Shamir, *The discrete log is very discreet*, Proc. of the 22nd Annual ACM Symposium on Theory of Computing, Baltimore, MD, 1990, Association for Computing Machinery, New York, 405–415.
61. Daniel Shanks, *Class number, a theory of factorization, and genera*, Proc. Symposium Pure Mathematics, American Mathematical Society, 1972.
62. Samuel S. Wagstaff, Jr., *Greatest of the least primes in arithmetic progressions having a given modulus*, Math. Comp. **33** (1979), 1073–1080.
63. A. E. Western and J. C. P. Miller, *Tables of indices and primitive roots*, Royal Society Mathematical Tables, vol. 9, Cambridge University Press, 1968.
64. Douglas H. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Trans. Inform. Theory **32** (1986), 54–62.
65. M. V. Wilkes, *Time-sharing computer systems*, American Elsevier, New York, 1968.

SANDIA NATIONAL LABORATORIES, ALBUQUERQUE, NM 87185