Computation of Discrete Logarithms in Fields of Characteristic Two

Daniel M. Gordon and Kevin S. McCurley
Sandia National Laboratories

July 2, 1991

1 Introduction

The difficulty of computing discrete logarithms was first proposed as the basis of security for cryptographic algorithms in the seminal paper of Diffie and Hellman [5]. The discrete logarithm problem in a finite group is the following: given group elements g and a, find an integer x such that $g^x = a$. We shall write $x = \mathrm{ind}_g a$, keeping in mind that $\mathrm{ind}_g a$ is only determined modulo the multiplicative order of g. For general information on the discrete logarithm problem and its cryptographic applications, the reader may consult [8] and [9]. In this paper we shall report on some computations done for calculating discrete logarithms in the multiplicative group of a finite field $\mathrm{GF}(2^n)$.

The computations that we carried out used a massively parallel implementation of Coppersmith's algorithm [3],which we outline in section 3. The results of our calculations will be presented in section 5. A great deal of effort (and CPU time!) has been expended on the cryptographically relevant problem of factoring integers, but rather little effort has gone into implementing discrete logarithm algorithms. The only published reports on computations using Coppersmith's algorithm are in [3, 4] and [1]. Both papers report on the calculation of discrete logarithms in the field $\mathrm{GF}(2^{127})$.

Odlyzko [9] has carried out an extensive analysis on Coppersmith's algorithm and projected the number of 32 bit operations required to deal with a field of a given size. From this analysis he made the prediction that fields

of size 2^{500} should be feasible using a supercomputer.A similar analysis was made by van Oorschot [10], with somewhat more OPTIMISTIC (?) predictions. Both of their predictions are consistent with our experience, and in particular we report here on the successful computation of discrete logarithms for fields of size up to GF(2^{503}). The major limitation at this point seems to lie as much in the linear algebra, which consumes not only a large amount of computation time but also requires large amounts of storage.

2 Coppersmith's algorithm

Coppersmith's algorithm belongs to a class of algorithms that are usually referred to as index calculus methods, and has three stages. In the first stage, we collect a system of linear equations (called relations) that are satisfied by the discrete logarithms of certain group elements belonging to a set called a factor base. In our case, the equations are really congruences modulo the order of the group, or modulo $2^n - 1$. In the second stage, we solve the set of equations to determine the discrete logarithms of the elements of our factor base. In the third stage, we compute any desired logarithm from our precomputed library of logarithms for the factor base.

For the Coppersmith algorithm, it is convenient that we construct our finite field GF(2^n) as GF(2)[x]/($f(x)$), where f is an irreducible polynomial of the form $x^n + f_1(x)$, with f_1 of small degree. Heuristic arguments suggest that this should be possible, and a search that we made confirms this, since it is possible to find an f_1 of degree at most 11 for all n up to 600, and it it is usually possible to find one of degree at most 7. For the construction of fields, it is also convenient to choose f so that the element $x \pmod{f(x)}$ is primitive, i.e. of multiplicative order $2^n - 1$.

For a given polynomial f that describes the field, there is an obvious projection from elements of the field to the set of polynomials over GF(2) of degree at most n. In our case, we shall take as our factor base the set of field elements that correspond to the irreducible polynomials of degree at most B for some integer B to be determined later. Let m be the cardinality of the factor base, and write g_i for an element of the factor base. We note that an equation of the form

$$\prod_{i=1}^{m} g_i^{e_i} = x^t$$

implies a linear relationship of the form

$$\sum_{i=1}^{m} e_i \text{ind}_x g_i \equiv t \pmod{2^n - 1}.$$

In order to describe the first stage in the Coppersmith method, we shall require further notation. Let r be an integer, and define $h = \lfloor n2^{-r} \rfloor + 1$. To generate a relation, we first choose random relatively prime polynomials $u_1(x)$ and $u_2(x)$ of degrees $\leq d_1$ and d_2 respectively. We then set $w_1(x) = u_1(x)x^h + u_2(x)$ and

$$w_2(x) = w_1(x)^{2^r} \pmod{f(x)}. \tag{1}$$

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

$$w_2(x) = u_1(x^{2^r})x^{h2^r - n} f_1(x) + u_2(x^{2^r}), \tag{2}$$

so that $\deg(w_2) \leq \max(2^r d_1 + h2^r - n + \deg(f_1), 2^r d_2)$. If we choose d_1, d_2, and 2^r to be of order $n^{1/3}$, then the degrees of w_1 and w_2 will be of order $n^{2/3}$. 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 B. If so, then from (1) we obtain a linear equation involving the logarithms of polynomials of degree $\leq B$.

An asymptotic analysis of the algorithm suggests that it is possible to choose the parameters so that the asymptotic running time of the first stage of the algorithm is of the form in such a way that the expected running time to complete stage one is of the form

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

In the second stage, the solution of linear equations modulo $2^n - 1$ can be accomplished using fairly well understood algorithms. Since most linear algebra algorithms are designed to work over a field, it is convenient to work modulo the prime factors of $2^n - 1$ and use Hensel's lemma and the chinese remainder theorem to produce the solutions modulo $2^n - 1$ (factorizations of $2^n - 1$ can be found in [2]). Ordinary Gaussian elimination takes about $m^3/3$ operations (modulo some prime factor p), but this analysis does not take into account the fact that the equations produced in stage 1 are extremely sparse.

For sparse equations there are several methods that can be employed, and in theory we can solve the systems in time $O(B^{2+\epsilon})$ operations.

An analysis of the running time for the third stage (which we do not describe in detail here) suggest a running time of

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

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

The preceding statements pertain to the asymptotic running time, but

2.1 Refinements of Stage 1.

Odlyzko has suggested several ways to speed up the performance of stage 1. None of these affect the asymptotic running time, but each of them may have some practical significance by speeding up the implementation by a factor of two or three. We shall not discuss these methods in great detail, but merely report on which of the methods we chose to implement.

2.1.1 Forcing a Factor Into w_1 and w_2

One method that was suggested by Odlyzko for improving the probability that w_1 and w_2 were smooth was by forcing them to contain at least one small degree factor. The method is described in complete detail in [9] and [10], but roughly speaking we fix polynomials v_1 and v_2 of degree at most B, and consider those (u_1, u_2) pairs for which w_1 and w_2 are divisible by v_1 and v_2 respectively. The (u_1, u_2) pairs with this property are described by a rather small set of linear equations modulo 2, and we can easily find such pairs by Gaussian elimination. For the size fields that we considered, the linear systems had fewer than 50 rows and equations, and a special purpose routine to solve these systems proved to be extremely efficient (rows could be added together by using two xor operations). One problem with this method is different v_1, v_2 pairs can lead to the same u_1, u_2 pairs, making it rather difficult to avoid duplication of effort. As far as we can tell, we were the first to implement this method, and our experience with it seemed to agree with the predictions made by Odlyzko.

2.1.2 Large Prime Variation

One well known method for speeding up the generation of equations is to use also equations that involve only one irreducible polynomial of degree only slightly larger than B. The rationale for this is that these equations can be discovered essentially for free, and two such equations involving the same "large prime" can be combined to produce an equation involving only the irreducibles of degree at most B. Many such equations can be discovered by checking whether after removing the smooth part from a polynomial, the residual factor has small degree. After combining two such equations, the equations produced are on average twice as dense as the other equations, so they complicate the linear algebra in stage 2. We chose to report the equations, but because of problems with the linear algebra, we have not yet made use of the extra equations.

2.1.3 Double Large Prime Variation

Just as we can use equations involving only a single irreducible of degree slightly larger than B, we can also use equations having two "large prime" factors. This has been used to speed up the quadratic sieve integer factoring algorithm [7], and we might expect the same sort of benefit when it is applied to the Coppersmith algorithm. Many such equations can be produced from reporting those u_1, u_2 pairs that produced a w_1 and w_2 both of which contained a large prime factor. Once again, we chose to report such equations, but have not been able to use them yet because of the difficulty in solving the (denser) system of equations.

2.1.4 Smoothness Testing

The most time-consuming part of the Coppersmith algorithm is the testing of polynomials for smoothness. At least two methods have been suggested for doing this, both of which are outlined in [9]. Of the two methods, we found the one used by Coppersmith to work faster for our implementation, and this was initially what we used.

After having carried out the computation for the case $n = 313$, we looked around for any variations that would speed up the smoothness testing. Drawing on the knowledge that sieving can be exploited to great advantage in integer factoring algorithms, we sought a way to use sieving to test many

polynomials simultaneously for smoothness. Sieving over the integers is relatively efficient due to the fact that integers that belong to a fixed residue class modulo a prime lie a fixed distance apart, and it is very easy to increment a counter by this quantity and perform a calculation on some memory location corresponding to the set element. For polynomials, the problem is slightly different, since we saw no obvious way of representing polynomials in such a way that representatives of a given residue class are a fixed distance apart. It turns out that this is not a great deterrent, since what is important is the ability to quickly move through the representatives, and for the data structures that we used, this can be done using the notion of a Gray code.

The method that we chose to use was to take a fixed u_1, and set up a segment of memory corresponding to the polynomials u_2 of degree up to d_2. This memory segment was initially set to contain all zeros. Then for each irreducible polynomial q of degree $d \leq B$, we stepped through those memory locations corresponding to the w_1's that are divisible by q, incrementing the memory location by d. After doing this for all irreducible powers of degree up to B, we scanned the memory segment to find those that contained a contribution large enough to show that the corresponding w_1 was built up from only small degree irreducibles. We could as easily have set up the sieving to be over polynomials w_2, but we chose to sieve over the w_1's because they always had larger degree and were therefore less likely to be smooth.

The reason that sieving works so well for the quadratic sieve algorithm is that it replaces multiple precision integer calculations with simple addition operations. We gain the same sort of advantage in Coppersmith's algorithm, by eliminating the need for many modular multiplications involving polynomials. The actual operation counts for sieving come out rather close to the operation counts given in [9] and [10], but in the case of sieving the operations are somewhat simpler.

2.1.5 Early Abort Strategy

One strategy that has been suggested for locating smooth integers is to search through random integers, initially dividing by small primes. At a certain point, we then check to see if the residual factor has moderate size, and abort the testing if it fails. It so happens that a random integer is more likely to be B-smooth from having many very small prime factors than it is from having just a few factors near B, and it follows that we should not

spend a lot of time dividing by moderately large primes to test for smoothness. This strategy has come to be known as the "early abort" strategy, and the same heuristic reasoning carries over to the smoothness testing part of Coppersmith's algorithm. Odlyzko predicted that this may result in a speedup of a factor of two in the algorithm, but we never got around to implementing it. The major reason for this is that there seems to be no obvious way to combine this idea with sieving, and the latter gave a somewhat better speedup.

2.1.6 Alternative Equations

In going through the u_1, u_2 pairs in a range of interest, those of small degree have a slight advantage in producing relations, for the simple reason that they lead to smaller degree w_1 and w_2 that are slightly more likely to be smooth. By employing a slight variation of (2), we can produce a different set of equations and effectively reduce the degree of u_1 or u_2 by one. For example, we can choose $h = \lfloor n2^{-r} \rfloor$ (smaller by one) and take $w_2 = x^{n-h2^r} w_1^{2^r}$. In this case, we get

$$w_2(x) = u_1(x^{2^r}) f_1(x) + x^{n-h2^r} u_2(x^{2^r}). \tag{3}$$

The equations generated in this manner seem to be independent from those produced by (2), and essentially allow us to reuse the same range of u_1 and u_2. This gives an expected speedup by a factor of about two.

2.2 Linear Algebra

The solution of sparse linear systems over finite fields have received much less attention than the corresponding problem of solving sparse linear systems over the field of real numbers. The fundamental difference between these two problems is that issues involving numerical stability problems arising from finite precision arithmetic do not arise when working over a finite field. The only pivoting that is required is to avoid division by zero. Algorithms for the solution of sparse linear systems over finite fields include:

- standard Gaussian elimination.
- structured Gaussian elimination.
- Wiedemann's algorithm.

- Conjugate Gradient.
- Lanczos methods.

A description of these methods can be found in the paper by LaMacchia and Odlyzko [6], where they describe their experience in solving systems that arise from integer factoring algorithms and the computation of discrete logarithms over fields $GF(p)$ for a prime p. We chose to implement two of these algorithms: conjugate gradient and structured Gaussian elimination. For handling multiple precision integers we used Lenstra-Manasse package. The original systems were reduced in size using the structured Gaussian elimination algorithm, after which the conjugate gradient algorithm was applied to solve the smaller (and still fairly sparse) system. All of the calculations were done on Sun workstations. For the case of $n = 313$, we chose a factor base of all polynomials of degree ≤ 21 (58636 of them). We generated approximately 84000 relations on these logarithms, which the structured Gaussian elimination program reduced down to a system of approximately 8500 equations in the same number of variables. This reduction of the system took only a few minutes on a Sun workstation. We then solved the smaller system using conjugate gradient algorithm, solving the system modulo p for the four prime factors p of $2^3 13 - 1$ (these were run in parallel on four workstations). The longest of these took approximately 7 days to run. After this we combined the solutions modulo p to get the solution modulo $2^3 13 - 1$ of the reduced system, and substituted these logarithms back into the original system to solve for most of the remaining logarithms of factor base elements. BOY IS THIS VAGUE.

3 Our Implementation

As Odlyzko pointed out in 1984, Coppersmith's algorithm parallelizes in a trivial way, by splitting up the testing of u_1, u_2 pairs across many processors. Sandia National Laboratories has two fairly large massively parallel systems in place at this time, namely a 1024 processor Ncube-2 MIMD "hypercube" and a 16384 processor Thinking Machines CM-2. When we started this project, it was not at all clear to us which of these machines would be better suited to Coppersmith's algorithm. There were at least two reasons that we originally thought the CM-2 might be better. The first reason is the fact that

it has more processors, even though they are considerably less powerful. The second reason is that the CM-2 uses bit-serial processors that are seemingly as happy dealing with bit fields of length 128 as they are with fields of length 32. We also perceived the CM-2 to provide a fairly nice interface for programming.

In examining the Coppersmith algorithm, it looks like a perfect example of an algorithm that can easily be implemented in the data-parallel paradigm of CM-2 programming. We planned to simply spread the u_1, u_2 pairs across the processors, and have them all perform the same smoothness test on their own w_1 and w_2. What we overlooked was the fact that the CM-2 must perform the same arithmetic or logical operation on all processors *using data located at the exact same memory address in each processor.* For our application, this was a serious problem. As an example of why, consider the calculation of polynomial gcd's, which requires repeated shifting and xoring of bit fields. Using any of the high-level languages on the CM-2, we could not see a way to shift fields by differing amounts corresponding to the data located on that processor. The only way to do this was to shift all polynomials by one position, and carry this out the addition on those processors that had their polynomials lined up correctly. This means that the worst case among the 16384 processors would determine the rate at which we could perform polynomial operations, and resulted in rather disappointing runtimes. While the CM-2 looks like a great bit-twiddling machine, we could not seem to efficiently do the kind of twiddling we needed to do!

By contrast, the processors on the Ncube-2 are rather standard 64 bit microprocessors that operate at approximately 4 MIPS. There are only 1024 of them, but this is more than made up for by the fact that it is able to xor fields of length 64 together in a single operation (in fact, due to a limitation in the current C compiler, we only used the 32 bit operations of the processors). For this machine we first wrote a serial version in standard C, and then simply added a few lines to instruct the processors as to what range of u_1 and u_2 they were to examine. We defined a single data structure called a `poly` whose members include an `int` to store the degree of the polynomial and an array of `unsigned int`'s to store the coefficients, packed 32 coefficients to a word. This packing was chosen not to save space, but rather to allow us to add polynomials by xoring the corresponding `int`'s together.

We were hindered at several points along the way from the fact that the Ncube-2 was a research machine rather than a well-tested production

machine. Such machines are routinely used for applications in scientific computing, but not very often for the sort of work that we were doing. As a result, we were puzzled for some time by a problem that turned out to be a C compiler error involving the way shifts were done. The operating system for the individual processors also did not reliably support multitasking, and corruption of individual nodes required the entire machine to be rebooted (a process that takes only a few seconds, but is extremely disruptive when several users are using different subcubes).

The Ncube is a very powerful machine, but we also required a tremendous number of operations to be performed. Luckily the Coppersmith algorithm can be broken down into very small pieces that are independent, which allowed us to plan the calculation as a sequence of many batch jobs. We wrote a few short programs to poll the machine every few minutes, and load in new jobs whenever there was extra space. We also wrote a program to allow other users to kick us out from parts of the machine, so that it would not interfere with other users' work. This crude approach allowed us to get around the lack of multitasking on the processors, and obtain a large number of CPU cycles. We tended to use about one fourth of the processors during the day, and a large fraction of the whole machine at night. This lasted for several months while we completed the different cases described in section 5.

4 Results

We started out by repeating Coppersmith's calculation of discrete logarithms for GF(2^{127}). Our original goal was to determine whether it was possible to compute discrete logarithms for the field GF(2^{593}), which has been suggested for possible use in at least one existing cryptosystem. Odlyzko predicted that fields of size up to 521 should be tractable using the fastest computers available within a few years (exact predictions are difficult to make without actually carrying out an implementation). His predictions turned out to be fairly accurate, and it now appears to be within the realm of possibility to carry out the calculation for fields of this approximate size. We believe that 521 should now be possible to complete, albeit with the consumption of massive amounts of computing time.

We have actually completed most of the calculations required to compute discrete logarithms for the fields $GF(2^n)$ for $n = 227$, $n = 313$, and $n = 401$.

We have also started gathering relations for the case $n = 503$, and it appears that this will be possible within about six months elapsed time. We have confined ourselves to relatively small factor bases, primarily because of the difficulties we had in solving the linear systems. Completion of the case $n = 503$ will require significantly more effort to be spent on the linear algebra, and in particular we plan to implement the linear algebra algorithms on the Ncube (a Cray would probably suffice, but we do not have access to free time on such a machine).

Dan: We need to give some figures on the number of processor-hours used to generate the relations, and say more about the size of factor bases.

References

[1] I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone. Computing logarithms in fields of characteristic two. *SIAM Journal of Algebraic and Discrete Methods*, 5:276–285, 1984.

[2] John Brillhart, D. H. Lehmer, J. L. Selfridge, Bryant Tuckerman, and Jr. S. S. Wagstaff. *Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers*, volume 22 of *Contemporary Mathematics*. American Mathematical Society, Providence, second edition, 1988.

[3] D. Coppersmith. Fast evaluation of discrete logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, 30:587–594, 1984.

[4] D. Coppersmith and J. H. Davenport. An application of factoring. *Journal of Symbolic Computation*, 1:241–243, 1985.

[5] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:472–492, 1976.

[6] B. A. LaMacchia and A. M. Odlyzko. Solving large sparse linear systems over finite fields. preprint, 1990.

[7] A. K. Lenstra and Mark Manasse. Factoring with two primes.

[8] Kevin S. McCurley. *The Discrete Logarithm Problem*, volume 42 of *Proceedings of Symposia in Applied Mathematics*, pages 49–74. American Mathematical Society, Providence, 1990.

[9] A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Advances in Cryptology (Proceedings of Eurocrypt 84)*, number 209 in Lecture Notes in Computer Science, pages 224–314, Berlin, 1985. Springer-Verlag.

[10] Paul C. van Oorschot. A comparison of practical public-key cryptosystems based on integer factorization and discrete logarithms. preprint, 1991.