Fast Key Exchange with Elliptic Curve Systems

Richard Schroeppel Hilarie Orman Sean O'Malley Oliver Spatscheck *
{rcs, ho, sean, spatsch}@cs.arizona.edu
Department of Computer Science, University of Arizona

Abstract. The Diffie-Hellman key exchange algorithm can be implemented using the group of points on an elliptic curve over the field $\mathbb{F}_{2^n}$. A software version of this using $n = 155$ can be optimized to achieve computation rates that are slightly faster than non-elliptic curve versions with a similar level of security. The fast computation of reciprocals in $\mathbb{F}_{2^n}$ is the key to the highly efficient implementation described here.

1 Introduction

The Diffie-Hellman key exchange algorithm [11] is a method for initiating an encrypted conversation between two previously unintroduced parties. It relies on exponentiation in a large group, and the software implementation of the group operation is usually computationally intensive. The algorithm has been proposed as an Internet standard [15], and the benefit of an efficient implementation would be that it could be widely deployed across a variety of platforms, greatly enhancing the security of the Internet by solving the problem of key exchange for millions of host machines.

The Diffie-Hellman algorithm was implemented several years ago as part of the Sun SecureRPC system used by Sun Microsystems, and the implementation used numbers of a size that was determined in [21] to be attackable using a method described in [9]. This work indicated that instead of using a 192 bit modulus, which could be "cracked" in only about 3 months of effort (including software development), system designers should use at least a 512 bit modulus. Informal conversations with people associated with developing the Sun SecureRPC system indicated that they did not wish to increase the size of the numbers, in part because of the amount of time needed for the computation. The extra time results because of the number of large-number arithmetic operations that must be carried out.

In our work with implementations of cryptographic protocols, we developed a simple version of the Diffie-Hellman protocol in what has been termed *transient* mode, where the two parties each select a random exponent e and exchange values of g^e, where g is a group element. If party A selects $e = a$ and party B selects $e = b$, then each party can compute g^{ab}, but no eavesdropper can do so. In our implementation, we used the ring $\mathbb{Z}_p$, with p a 512 bit prime, a size that should resist attacks with hardware resources known today. The protocol took from 2 to 10 seconds on a variety of modern and popular hardware platforms.

* This work was supported by the National Computer Security Center, contract MDA904-92-C-5151.

D. Coppersmith (Ed.): Advances in Cryptology - CRYPTO '95, LNCS 963, pp. 43-56, 1995.

This speed is unpalatable for machines that need to participate in many keyed conversations with a large set of peers. We would estimate that no busy machine should devote more than .1% of its cycles to key exchange, and this limits even a very fast machine ($\leq$ 200 MHz) to fewer than 20 key exchanges per hour.

This work motivated our research into faster software implementations of the basic operations behind the protocol. Elliptic curve systems, first suggested by Victor Miller [25] and Neal Koblitz [19], were a natural choice because they are (insofar as is known today) immune to the index calculus attack. This means that smaller numbers can be used to achieve the same degree of security for the Diffie-Hellman algorithm as the 512 bit version described above. It is interesting to note that the numbers in our implementation are even smaller than the Sun RPC version. In addition to this basic savings in computation cost, there are several software optimization techniques that result in a significantly faster algorithm.

An additional advantage is that, as computers get faster, the size of the numbers needed to achieve a particular level of security grows much more slowly for elliptic curve systems when compared to methods that use ordinary integers.

The elliptic curve method uses a different group operation than multiplication of integers mod p. Instead, the operation is over the group of points on an elliptic curve, and the operation is arithmetically more complicated. The size of the group used in our implementation is approximately 2^{155}. The group operation is implemented using numbers from the Galois field $\mathbf{F}_{2^{155}}$. Our initial implementation of this was more than twice as fast as the implementation using integers modulo a 512 bit prime, and there was obvious room for improvement. For the DH key exchange algorithm, a properly chosen elliptic curve over $\mathbf{F}_{2^{155}}$ offers somewhat more security than does working modulo a 512 bit prime.

The improvements described here are how to efficiently compute the field operations in $\mathbf{F}_{2^{155}}$, especially reciprocals, and a minor improvement in the formula for doubling an elliptic curve point. The most important contributor to the success of the algorithm is the fast reciprocal routine.

2 Overview of the Method

We include here brief descriptions of the field and elliptic curve manipulations; this material is from draft document [24]. See Silverman [32] for a general introduction to elliptic curves; Menezes [23] provides a cookbook approach and an introduction to the cryptographic methods. Other good references are [1, 2, 3, 5].

For our purposes, an elliptic curve E is a set of points (x, y) with coordinates x and y lying in the field $\mathbf{F}_{2^{155}}$ and satisfying a certain cubic equation. The points (x, y) form a commutative group under "addition"; the rule for "addition" involves several field operations, including computing a reciprocal; the formulas are in section 3.1.

The elliptic curve analog of the Diffie-Hellman key exchange method uses an elliptic curve E, and a point $P = (x_0, y_0)$ which generates the whole addition group of E. The base field, curve equation, and starting point are all system-wide public parameters.

When user A wants to start a conversation, he chooses a secret integer multiplier K_A in the range $[2, order(E) - 2]$ and computes K_AP by iterating the addition of P using a "double and add" scheme. He then sends the x and y coordinates of the point K_AP to user B. User B selects his own secret multiplier K_B and computes and sends to user A the point K_BP. Each user can then compute the point $(K_AK_B)P$. Some bits are selected from the coordinates to become the secret session key for their conversation. Insofar as is known, there is no effective method for recovering $(K_AK_B)P$ by eavesdropping on this conversation, other than solving the discrete logarithm problem. The discrete logarithm problem is hard for elliptic curves, because the index calculus attack that is so effective modulo p does not work.

The elliptic curve operations require addition, multiplication, squaring, and inversion in the underlying field. The number of applications of each operation depends on the exact details of the implementation; in all implementations the inversion operation is by far the most expensive (by a factor of 5 to 20 over multiplication).

3 Working with an Elliptic Curve

3.1 Adding and Doubling Points

The two elliptic curve operations that are most relevant to the complexity of multiplying a group element by a constant are the Add and Double operations. They are presented slightly modified from their presentation in [23]. The elliptic curve $E_{a,b}$ is the set of all solutions (x, y) to the equation $y^2 + xy = x^3 + ax^2 + b$. Here a and b are constants from the field $\mathbb{F}_{2^{155}}$; b must be nonzero. The variables x and y are also elements of $\mathbb{F}_{2^{155}}$. A solution (x, y) is called a *point* of the curve. An extra point O is used as the group identity. We use (0,0) to represent O. (Because $b \neq 0$, (0,0) is never a solution of our equation.)

(i) Adding two points (x_1, y_1) and (x_2, y_2):
If either point is O, return the other point as the sum.
If $x_1 = x_2$: When $y_1 \neq y_2$, return O; otherwise use the Doubling rule.
If $x_1 \neq x_2$, then $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$, where

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a, \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1, \quad \text{and} \quad \lambda = \frac{y_1 + y_2}{x_1 + x_2}.$$

(ii) Doubling a point (x, y):
If $x = 0$, then $2(x, y) = O$. Otherwise, $2(x, y) = (x_2, y_2)$, where

$$x_2 = \lambda^2 + \lambda + a, \quad y_2 = x^2 + (\lambda + 1)\, x_2, \quad \text{and} \quad \lambda = \left(x + \frac{y}{x}\right).$$

x_2 as given in [23] uses an extra multiplication. Our formula can be further improved to remove one multiplication [31].

(iii) Negating a point (x, y): $\quad -1(x, y) = (x, x + y)$.

From these formulas, we can determine the number of field operations required for each kind of elliptic curve operation. An Addition step usually requires eight additions, two multiplications, one squaring, three reductions modulo the field polynomial $T(u)$, and one inversion. A Doubling step usually requires four additions, two multiplications, two squarings, four reductions mod $T(u)$, and one inversion. A Negation step requires one addition. The important contributors to the running time are the multiplications and inversions.

3.2 Choosing the Curve

The constant a in the elliptic curve equation can be chosen to simplify the operations of doubling a point and of adding two points. We use $a = 0$, which eliminates one addition from the formula for the x coordinate in both operations.

The order of the group is roughly 2^{155}, but the exact size depends on the choice of a and b. There is a complicated algorithm due to Schoof [30], with improvements by Atkin, Elkies, Morain, and Couveignes [10] for determining the group order. For maximum security, the order should have as large a prime factor as possible. In our equation, with $a = 0$, the best possible order is $4p$, with p a prime near 2^{153} [20]. (If $a \neq 0$, the order can have the form $2p$ with p near 2^{154}, giving a bit of extra security.) Lay and Zimmer [22] give a method for creating a curve with a given order, but we are reluctant to use their scheme because it produces curves closely related to *rational* curves with an extra structural property called *complex multiplication*. We don't know of any way to "crack" such curves, but it seems prudent to avoid this extra structure.

A curve is selected by choosing small values for (x, y) and computing b from the equation $y^2 + xy = x^3 + b$. The curve order is computed with Schoof's algorithm, and tested to see if it is of the form $4p$. For curves based on $\mathbb{F}_{2^{155}}$, a few hundred tries may be necessary.

The best known methods for computing elliptic curve discrete logarithms take time proportional to the square root of the largest prime factor of the group order [27, 28, 14]. In our case, the largest prime factor will be about 2^{153}, so finding discrete logarithms will take about $2^{76.5} \approx 10^{23}$ operations.

3.3 Computing a Multiple of a Point

The number of additions and doublings necessary for computing nP (where P is a point on the curve and n is an integer) is an important factor in the speed of the key exchange algorithm. We initially implemented the straightforward double-and-add approach based on the binary expansion of n. For a random 155 bit multiplier n, computing nP requires 154 doubling steps and an average of 77 addition steps. The number of doubling steps is roughly fixed, but it is possible to reduce the number of addition steps. This problem is studied in a large literature on *addition chains* [18, 7, 8, 29]. We implemented two well-known easy speedups that apply to randomly chosen multipliers.

If the starting point P is available ahead of time, preparation of tables of multiples of P is useful [8]. This is the situation for the first two of the four

point multiplications in key exchange where both parties begin with the system-wide generator $P = (x_0, y_0)$. The precomputed table consists of $16^K P$ for $K = 0, \ldots, 38$. Using radix 16 and the digits ± 1, a typical 155 bit multiplier n requires about 42 point additions (and no doublings) to compute nP.

When the starting point P is not known ahead of time, as for the final two key exchange point multiplications, a different speedup is available. This is a blend of the *m-ary* method ([18] p. 404) with Booth's algorithm [6]. In this case, the computed table is the odd multiples from P to $15P$. Then we proceed as in the usual double-and-add method, scanning the bit representation of the multiplier n from the high end. Using the table of small multiples of P, we can do several doubling steps before an addition is necessary. If we require an even multiple of P, such as $12P$, we instead use an odd multiple ($3P$), introduced a couple of steps earlier in the doubling process. Because the subtraction of two curve points is no costlier than addition, we have the option of subtracting a small multiple of P when convenient. On the average, the number of addition/subtraction steps needed is $\frac{1}{6}$ the size of the exponent. For a random 155 bit exponent, we will use about 152 doubling steps and 32 addition/subtractions (including the cost of preparing the table).

For DH key exchange, the first two of the point multiplications are with a known starting point, and the last two are with new points. The last two point multiplications can take place in parallel. The total time for the key exchange is $2(42A) + (152D + 32A) = 152D + 116A$. The double-and-add approach would use $3(154D + 77A) = 462D + 231A$, nearly three times as many operations.

4 Field Operations in $\mathbb{F}_{2^{155}}$

4.1 Representation of the Field Elements

We represent field elements as bitstrings of length 155. For a 64 bit processor, this is only 3 words; the brevity of the representation means that much of the computation can be done in hardware registers.

Let $k[u]$ be the ring of polynomials over $\mathbb{F}_2$. We will work in the extension field of the trinomial $T(u) = u^{155} + u^{62} + 1$. The extension is a field because the polynomial is irreducible over $\mathbb{F}_2$. The field elements are members of $k[u]$ modulo the field polynomial $T(u)$, with coefficients drawn from the set $0, 1$. Each polynomial in $k[u]$ can be reduced to a remainder of degree at most 154.

The irreducible trinomial $T(u)$ has a structure that makes it a pleasant choice for representing the field. In $\mathbb{F}_2$, there are only two irreducible polynomials of this degree. The fact that the middle term, u^{62}, has an exponent that is roughly half of the field degree is important to the optimizations for calculating modular reductions (as described in section 4.3) and to the division by large powers of u, (as explained in section 4.4).

4.2 Addition and Multiplication

Field elements (for a prime power field) are added and multiplied as follows:

- *Field addition:* $(a_{n-1} \cdots a_1 a_0) + (b_{n-1} \cdots b_1 b_0) = (c_{n-1} \cdots c_1 c_0)$, where $c_i = a_i + b_i$ in the field $\mathbb{F}_2$. That is, field addition is performed componentwise.
- *Field multiplication:* $(a_{n-1} \cdots a_1 a_0) \cdot (b_{n-1} \cdots b_1 b_0) = (r_{n-1} \cdots r_1 r_0)$, where the polynomial $(r_{n-1}u^{n-1} + \cdots + r_1 u + r_0)$ is the remainder when the polynomial $(a_{n-1}u^{n-1} + \cdots + a_1 u + a_0) \cdot (b_{n-1}u^{n-1} + \cdots + b_1 u + b_0)$ is divided by $T(u)$ over $\mathbb{F}_2$.

The addition algorithm for field elements is trivial: the two blocks of bits are simply combined with the bitwise *xor* operation. Because our field has characteristic 2, subtraction is the same as addition, and negation is the identity operation.

Multiplication of field elements uses the same shift-and-add algorithm as is used for multiplication of integers, except that the "add" is replaced with "xor". This has the virtue that the operation can no longer generate carries, simplifying the implementation. We experimented with several different ways of organizing the multiplication routines and found that different architectures had different optimal routines. (Our timings are done with the optimal routines for each architecture.)

The Squaring Operation Squaring a polynomial in a modulo 2 field is a *linear* operation. In the formula for squaring a binomial, $(a+b)^2 = a^2 + 2ab + b^2$, the cross-term vanishes modulo 2 and the square reduces to $a^2 + b^2$. Consequently, we can square a sum by squaring the individual terms. For example, $(u^3 + u + 1)^2 = u^6 + u^2 + 1$.

In terms of bitstrings, to square a polynomial, we spread it out by interleaving a 0 bit between each polynomial bit. For example, $u^3 + u + 1$ is represented as **1011**, and the square is **1000101**. This can be done quickly using table lookup to convert each byte to its 15 bit square. The squared polynomial is then reduced modulo $T(u)$. Squaring is so much faster than regular multiplication that it can be ignored in rough comparisons of the timings.

4.3 Modular Reduction

The field elements are polynomials with coefficients in the ring $\mathbb{Z}_2$. After each multiplication or squaring, the result must be reduced modulo $T(u) = u^{155} + u^{62} + 1$. The product of two polynomials of degree 154 produces a polynomial of degree 308. The product is represented as 10 words on a 32 bit architecture, or 5 words on a 64 bit architecture.

A hand tailored reduction method, specific for $T(u)$, takes advantage of the degree of the middle term to minimize the number of operations required. First note that

$$u^{155} \equiv u^{62} + 1, \quad \text{and} \quad u^n \equiv u^{n-93} + u^{n-155} \bmod T(u).$$

Assume the polynomial to be reduced is

$$P(u) = a_{308}u^{308} + \cdots + a_2 u^2 + a_1 u + a_0.$$

As many as 93 of the leading terms of $P(u)$ can be reduced modulo $T(u)$ by replacing each non-zero term by its congruent two-term expression, i.e. $au^n \equiv au^{n-93} + au^{n-155} \bmod T(u)$. We can think of this as zeroing out the upper 93 bits of the 309 bits of the expression (subtracting each term) and adding in the representation of each original term right-shifted by 93 (i.e., multiplied by u^{-93}) and also right-shifted by 155 (i.e., multiplied by u^{-155}):

$$P(u) \equiv P(u)_{215-0} + P(u)_{308-216}(u^{-93} + u^{-155}) \bmod T(u)$$

where $P(u)_{j-k} = \sum_{i=j}^{k} a_i u^i$ is the portion of $P(u)$ from degrees j through k. This yields a length 216 partial result. This reduction can be repeated to make the degree less than 155.

In practice, we work one computer word at a time, lowering the degree by either 32 or 64, proceeding from the high order terms (bits) to the low. The results are accumulated into the original expression, i.e. the bitstring representing $P(u)$ is the operand for each shift and xor operation.

The benefit of using a trinomial as the modulus is that each word only needs to be xored into two places for the accumulation operation. Having the middle term of relatively low degree is beneficial because the accumulation operation with a high-order word does not affect that word, so that each reduction step reduces the degree by a full word. (If the middle term were u^{150} instead of u^{62}, we would only shorten our dividend by 5 bits each time instead of 32, and we would have to do the reduction operation multiple times.)

Some other recommended trinomials are $u^{127}+u^{63}+1$, $u^{140}+u^{65}+1$, $u^{172}+u^{81}+1$, $u^{185}+u^{69}+1$, $u^{191}+u^{71}+1$, $u^{217}+u^{64}+1$, $u^{223}+u^{91}+1$, and $u^{255}+u^{82}+1$. Irreducible trinomials are somewhat sparse: for the degrees from $100-199$, 43 have no irreducible trinomial. If one needs to work with a field of a specific degree, and the field has no good trinomial, a pentanomial (at least) is required. (In fields of characteristic 2, polynomials with an even number of terms are always divisible by $u+1$.)

4.4 Computing Reciprocals

The rules for doubling an elliptic curve point, and for adding two elliptic curve points, involve computing a reciprocal, either $1/x$ or $1/(x_1+x_2)$ (see section 3.1). Multiplicative inversion of elements in a field is usually so slow that people have gone to great lengths to avoid it. Menezes [23] (p. 90) and Beth and Schaefer [5] discuss projective schemes, which use about nine multiplications per elliptic curve step, but use very few reciprocals. We report here on a relatively fast algorithm for field inversion, which allows direct use of the simple formulas for operating on elliptic curve points. Our field inversion time is about three multiplication times, a substantial improvement over [23]. [2]

For the field we are working in, the problem to be solved is

[2] Menezes' inversion scheme computes $1/A(u)$ as $A(u)^{2^{155}-2} \bmod T(u)$. This can be done with 10 multiplications and 154 squarings [16].

Given a non-zero polynomial $A(u)$ of degree less than or equal to 154, find the (unique) polynomial $B(u)$ of degree less than or equal to 154 such that

$$A(u)B(u) \equiv 1 \bmod u^{155} + u^{62} + 1.$$

The problem has a simple, but relatively slow, recursive solution, exactly analogous to the related algorithm for integers. We have developed an algorithm that is considerably faster. It borrows ideas from Berlekamp [4] and from the low-end GCD algorithm of Roland Silver, John Terzian, and J. Stein (described in Knuth [18] p. 297). Our Almost Inverse algorithm computes $B(u)$ and k such that

$$AB \equiv u^k \bmod M, \quad deg(B) < deg(M), \quad \text{and} \quad k < 2deg(M),$$

where $deg(B)$ denotes the polynomial degree of B. After executing the algorithm, we will need to divide B by $u^k \pmod M$ to get the true reciprocal of A.

The pseudo-code for the algorithm is given below. The computer implementation relies on a few representational items:

- Multiplication of a polynomial by u is a left-shift by 1 bit.
- Division of a polynomial by u is a right-shift by 1 bit.
- A polynomial is **even** if its least significant bit, the coefficient of u^0 (the constant term), is 0. Otherwise it is **odd**.

The algorithm will work whenever $A(u)$ and $M(u)$ are relatively prime, $A(u) \neq 0$, $M(u)$ is odd, and $deg(M) > 0$. In our application, $M(u)$ is the irreducible trinomial $T(u)$, so the algorithm works for any nonzero $A(u)$ with degree less than $T(u)$.

The Almost Inverse Algorithm

```
        Initialize integer k=0, and polynomials B=1,C=0,F=A,G=M.

loop:   While F is even, do F=F/u, C=C*u, k=k+1.
        If F = 1, then return B,k.
        If deg(F) < deg(G), then exchange F,G and exchange B,C.
        F=F+G, B=B+C.
        Goto loop.
```

We improved the performance of this raw algorithm considerably with the following programming tricks:

- The operations on the polynomials B, C, F, G are made into inline, loop-unrolled code within the inversion routine. This is a crucial optimization, resulting in a factor of 3 reduction in the overall running time.
- Instead of using small arrays for B, C, F, G, use separate named variables $B0, B1, \ldots, G4$ etc. to hold the individual words of the polynomials. Assign as many of these as possible to registers.
- F is even at the bottom of the loop, so the "Goto" can skip over the test for the "While". This non-structured jump into the body of a "While" loop saves about 10% of the time.

- Instead of exchanging F, G and B, C, make two copies of the code, one with the names exchanged. Whenever an exchange would be called for, instead jump to the other copy.
- During the execution of the code, the lengths of the variables F, G shrink, while B, C grow. Detect when variables' lengths cross a word boundary, and switch to a copy of the code which knows the exact number of words required to hold the variables. This optimization makes the code much larger, because either 25 (for a 32 bit architecture) or 9 (for a 64 bit architecture) copies are required. Fortunately the code still fits within the DEC Alpha on-chip cache.

The following additional optimization is possible:

- Because F, G shrink while B, C expand, some of the variables representing the high-order terms can share a machine register. This would be useful on register-poor machines.

Dividing out u^k To find the reciprocal of $A(u)$, we need to divide $B(u)$ by u^k, working mod $T(u)$. The typical value of k is 260, although k can be as large as 309. The strategy is to divide B successively by u^w, where w is the number of bits in the wordsize of the computer, and finish up with a final division by a smaller power of u.

The operation of dividing by u^w is broken into two parts. First, a suitably chosen multiple of T is added to B, so as to zero out the w low order bits of B. The new B can have degree as large as $154 + w$. Second, the new B is right-shifted by w bits, effectively dividing it by u^w. Since the low order bits are 0, the division is exact; and the right-shift reduces the degree to (at most) 154.

The "suitably chosen multiple of T" is just T times the low order 32 (or 64) bits of B. For the 32 bit SPARC, using the notation of section 4.3,

$$B \equiv B + B_{31-0}(u^{155} + u^{62} + 1) \equiv B_{154-32} + B_{31-0}(u^{155} + u^{62}) \bmod T(u).$$

The second term is computed by left-shifting the low-order 32 bits of B by 62 bits and 155 bits and is xored directly into B. The zeroing operation has a complication on the Alpha, where we work with 64 bits at a time: After the shift-and-two-xors step, there are two possibly unzeroed bits, B_{63-62}. An additional shift-and-two-xors step is performed with this twenty-five cent field to clear it.

The same logic, modified for the smaller shift size, is used for the final division by a less-than-wordsize power of u.

5 Timings

The timings on two platforms are presented in Figure 1. The Sun SPARC IPC is a 25 MHz RISC architecture, with a 32 bit word size. The DEC Alpha 3000 is a 175 MHz RISC architecture, with a 64 bit word size. If everything is just right (all the data in registers, etc.), the SPARC machine can execute 25 million instructions per second, the Alpha 175 million. The Alpha has an 8K byte on-chip instruction cache; assuring that the critical field operations are loaded into

Field and Curve Operations	SPARC IPC	Alpha
155 bit add	3.5	.22
155 x 155 bit multiply	112.6	7.10
32 x 32 bit multiply	8.0	
64 x 64 bit multiply		1.65
155 bit square	8.1	.49
Modular reduction, 310 bits to 155 bits	3.8	.15
Reciprocal, including divide by u^k ($k = 261$)	280.1	25.21
Double an elliptic curve point	481.0	40.46
Add two elliptic curve points	550.8	42.05
Multiply known elliptic curve point	23 msec	1.8 msec
Multiply new elliptic curve point	92 msec	7.8 msec
Key exchange times		
Elliptic curve DH key exchange (155 bits)	137 msec	11.5 msec
Ordinary integer DH key exchange (512 bits)	2670 msec	185 msec
... with four tricks (estimate)	523 msec	34.8 msec
... no tricks, 128 bit key (estimate)	670 msec	46 msec
Best DH key exchange, estimates for 128 bit multiplier/exponent		
Elliptic curve over $\mathbb{F}_{2^{155}}$ (estimate)	114 msec	9.6 msec
Modular arithmetic, 512 bit prime (estimate)	150 msec	10.0 msec

Fig. 1. Times for various field and elliptic curve operations. Unlabeled times are in microseconds.

the cache without conflict is crucial to achieving the results reported here. The SPARC instruction cache is 64K bytes, and the code fits easily.

We made a few measurements on other architectures. The Intel 486 (66MHz) and the DEC MIPS (25MHz) are both within 10% of the SPARC times. Both machines have a 32 bit word size.

Assumptions for Timing Estimates Our modular arithmetic implementation of Diffie-Hellman key exchange uses the GNU bignum package GMP [13]. This package includes assembly language routines for the low-level primitives.

For a fair comparison with ordinary modular key exchange, we assume that all reasonable optimizations are made to the arithmetic. Many of the tricks that we have used in our elliptic curve program have analogs in ordinary arithmetic. For comparison purposes, we assume that the four tricks described below are added to GMP.

- Special case code for squaring a number is used to reduce the time to 60% of a multiplication.
- The modulus is chosen to be a prime P of the form $2^{512} - K$ with small K, reducing the time for a modular reduction to 10% of a multiplication.
- When the base g to be exponentiated is known ahead of time, we assume that tables of $g^{\pm 32^K} \pmod P$ are prepared in advance. For a random 512 bit

exponent, this will reduce the required number of modular multiplications to 114 on the average, and eliminate all squarings.
- When the base B is not known in advance, we assume the *m-ary* method [18], using the odd powers $B^1, \ldots, B^{31} (\mathrm{mod}\ P)$. For a random 512 bit exponent, this will require an average 508 modular squarings and 100 modular multiplications.

Under these assumptions, the estimated times for modular arithmetic DHKX are given in the table above. These times are compared with actual measured times for the implemented elliptic curve routines.

Phil Karn has suggested speeding up the key exchange by limiting the exponent size to 128 bits, in Photuris [15]. Therefore, we also provide estimates for both methods of key exchange on the assumption that the exponents/multipliers are reduced to 128 bits. In this case, the elliptic curve method has only a slightly better running time than the modular arithmetic method.

5.1 Planning Ahead

A major advantage of elliptic curves is that they scale well with increasing computer power. In the discrete logarithm problem for modular arithmetic, changing the modulus from a 512 bit prime to a 1024 bit prime multiplies the cracking effort by a factor of 8 million.[3] In the elliptic curve case, forcing a corresponding increase in cracking effort requires adding only 46 bits to the size of the field. This would raise our field size from 155 bits to 201 bits.

Now compare the changes in encryption effort for this increased security. In the ordinary arithmetic case, changing from a 512 bit modulus to 1024 bits will make the basic operation of modular multiplication take 3 times as long if the Karatsuba method [17] is used. The corresponding elliptic curve times increase by only $(201/155)^2 = 1.7$. (Both methods would also require that exponent/multiplier sizes increase by 46 bits, so each method incurs an additional penalty of about $174/128 = 1.4$.)

This advantage of elliptic curves is even more pronounced if long-term security is required: each additional bit added to the field size increases the cracking work by a factor of 1.4. In the modular arithmetic case, adding a bit to a 1024 bit prime only multiplies the required work by 1.026. To double the cracking effort requires increasing the the field size by 2 bits in the elliptic curve case, versus adding 27 bits to the length of the modulus for modular arithmetic.

6 Other Applications

ElGamal Encryption The elliptic curve improvements also make ElGamal style encryption and signatures more attractive. The total effort of signing and checking a signature is less with elliptic curve methods than with RSA.

[3] The time required to find a discrete logarithm mod p is estimated with the formula $\exp(1.93\sqrt[3]{\log p(\log\log p)^2})$ [21].

Cryptographic Operation	SPARC IPC	Alpha
Elliptic curve ElGamal encryption (155 bits)	116 msec	9.8 msec
Elliptic curve ElGamal decryption (155 bits)	94 msec	8.0 msec
Elliptic curve ElGamal signature (estimate)	24 msec	1.8 msec
Elliptic curve ElGamal check signature (estimate)	220 msec	17.8 msec

Fig. 2. Times for elliptic curve versions of ElGamal cryptographic operations.

The ElGamal encryption method [12, 23] can be implemented using these elliptic curve routines. The method uses a "semi-static" Diffie-Hellman key exchange: there is a public elliptic curve with generator $P = (x_0, y_0)$, and participants choose secret multipliers m_p and contribute $P_p = m_p P$ to a public key phone book. When party A wishes to communicate with party B, he selects a random multiplier k and computes kP and kP_B. The latter quantity is used to create a key for encrypting the message. The quantity kP is attached to the ciphertext. Party B can recover the key by computing $m_B kP$. Timings of the encryption and decryption operations are presented in Figure 2, along with estimates for the related signature operations.

Other Finite Fields The new reciprocal algorithm is useful for doing arithmetic in other finite fields. Because it makes inversion less costly, it will be worthwhile to reanalyze other formulas for operations with elliptic curves. The reciprocal algorithm can also be used, with slight modifications, to compute reciprocals in ordinary integer modular arithmetic. (The algorithm is most efficient with moduli of the form $2^A - 2^B - 1$, but will work reasonably with $2^A - k2^B - 1$ for 32 bit k. For generic odd moduli, Peter Montgomery's trick [26] is useful for dividing by the required power of 2.) Another benefit may be in ordinary modular exponentiation, as used in RSA and many other schemes: if a reciprocal costs only a few multiplications, addition-subtraction chains can be used to compute powers; this allows shorter chains which more than recoups the investment in computing the reciprocal.

7 Conclusions

We have shown that the software implementation of the Diffie-Hellman algorithm can be done more efficiently using elliptic curve systems over $\mathbb{F}_{2^n}$ than using integers modulo p. Assuming that no equivalent to the discrete logarithm attack exists for an elliptic curve, smaller number representations of the group elements can be used, and the software becomes quadratically faster. RISC machines with 64 bit wide words show excellent performance.

Our implementation's major speed advantage over previous implementations derives from its use of an efficient procedure for computing reciprocals in $\mathbb{F}_{2^n}$.

If network protocols were to rely on this method for establishing DES key pairs between hosts, four times as many connections could be made as compared with our baseline modulo p implementation. Even if the mod p arithmetic is improved as outlined in section 5, the elliptic curve method remains very competitive. As computing power increases, and the search capabilities of opponents improve accordingly, it is cheaper to improve the security of elliptic curve methods than to improve the security of mod p methods.

Key exchange nonetheless remains an expensive operation ... over 100 times as expensive as computing the MD5 one-way hash function, for example.

8 Acknowledgments

We thank R. W. Gosper for using MACSYMA to compute a table of factorizations of trinomials, and Alfred Menezes for providing us with reference [24].

References

1. G. AGNEW, T. BETH, R. MULLIN AND S. VANSTONE, "Arithmetic Operations in $GF(2^m)$", *Journal of Cryptology*, **6** (1993), 3-13.
2. G. AGNEW, R. MULLIN AND S. VANSTONE, "An Implementation of Elliptic Curve Cryptosystems over $F_{2^{155}}$", *IEEE Journal on Selected Areas in Communications*, **11** (1993), 804-813.
3. G. AGNEW, R. MULLIN, I. ONYSZCHUK AND S. VANSTONE, "An Implementation for a Fast Public-Key Cryptosystem", *Journal of Cryptology*, **3** (1991), 63-79.
4. ELWYN BERLEKAMP, *Algebraic Coding Theory*, McGraw-Hill, 1968, p.41.
5. T. BETH AND F. SCHAEFER, "Non Supersingular Elliptic Curves for Public Key Cryptosystems", *Advances in Cryptology – EUROCRYPT '91*, Lecture Notes in Computer Science, **547** (1991), Springer-Verlag, 316-327.
6. A. D. BOOTH, "A Signed Binary Multiplication Technique", *Q. J. Mech. Appl. Math.* **4** (1951), 236-240.
7. J. BOS AND M. COSTER, "Addition Chain Heuristics", *Advances in Cryptology – CRYPTO '89*, Lecture Notes in Computer Science, **435** (1990), Springer-Verlag, 400-407.
8. E. BRICKELL, D. GORDON, K. MCCURLEY, AND D. WILSON, "Fast Exponentiation with Precomputation (Extended Abstract)", *Advances in Cryptology – EUROCRYPT '92*, Lecture Notes in Computer Science, **658** (1993), Springer-Verlag, 200-207.
9. D. COPPERSMITH, A. ODLYZKO, AND R. SCHROEPPEL, "Discrete Logarithms in $GF[p]$", *Algorithmica*, **1** (1986), 1-15.
10. JEAN-MARC COUVEIGNES AND FRANÇOIS MORAIN *Algorithmic Number Theory: First International Symposium*, Lecture Notes in Computer Science, **877** (1994), Springer-Verlag, 43-58.
11. WHITFIELD DIFFIE AND M. E. HELLMAN, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, **IT-22**, n. 6, Nov. 1976, pp 644-654
12. T. ELGAMAL, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", *IEEE Trans. on Information Theory*, **31** (1985), 469-472.

13. TORBJORN GRANLUND, GMP, the GNU bignum package, version 1.3.2a, July 1994. ftp://prep.ai.mit.edu/pub/gnu/gmp-1.3.2.tar.gz
14. GREG HARPER, ALFRED MENEZES, AND SCOTT VANSTONE "Public-Key Cryptosystems with Very Small Key Lengths", *Advances in Cryptology – EUROCRYPT '92*, Lecture Notes in Computer Science, **658** (1993), Springer-Verlag, 163-173.
15. The Internet Engineering Task Force Working Group on Security for IPv4; drafts on key management available via FTP from the archives at `ds.internic.net` ; internet-drafts/draft-karn-photuris-00.txt
16. T. ITOH, O. TEECHI, AND S. TSUJII, "A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^t)$ Using Normal Bases" (in Japanese), *J. Society for Electronic Communications (Japan)*, 44 (1986), 31-36.
17. A. KARATSUBA, *Doklady Akademiia Nauk SSSR* **145** (1962), 293-294.
18. DONALD E. KNUTH, *Seminumerical Algorithms*, The Art of Computer Programming, **2** Addison Wesley 1969
19. NEAL KOBLITZ, "Elliptic Curve Cryptosystems", *Mathematics of Computation*, **48** n. 177 (1987), 203-209.
20. NEAL KOBLITZ, "Constructing Elliptic Curve Cryptosystems in Characteristic 2", *Advances in Cryptology – CRYPTO '90 Proceedings*, Lecture Notes in Computer Science, **537** (1991), Springer-Verlag, 156-167.
21. B. LA MACCHIA AND A. ODLYZKO, "Computation of Discrete Logarithms in Prime Fields", *Designs, Codes and Cryptography*, **1** (1991), p. 47-62.
22. G. LAY AND H. ZIMMER, "Constructing Elliptic Curves with Given Group Order over Large Finite Fields", *Algorithmic Number Theory: First International Symposium*, Lecture Notes in Computer Science, **877** (1994), Springer-Verlag, 250-263.
23. ALFRED J. MENEZES, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, 1993.
24. ALFRED J. MENEZES, MINGHUA QU, AND SCOTT A. VANSTONE, "Standard for RSA, Diffie-Hellman and Related Public Key Cryptography", Working Draft of IEEE P1363 Standard, April 24, 1995.
25. VICTOR S. MILLER, "Use of Elliptic Curves in Cryptography", *Advances in Cryptology – CRYPTO '85 Proceedings*, Lecture Notes in Computer Science, **218** (1986), Springer-Verlag, 417-426.
26. PETER L. MONTGOMERY, "Modular Multiplication without Trial Division", *Mathematics of Computation*, **44** (1985), 519-521.
27. P. VAN OORSCHOT AND M. WIENER, "Parallel Collision Search with Application to Hash Functions and Discrete Logarithms", 2nd ACM Conference on Computer and Communications Security, Fairfax, Virginia, November 4, 1994.
28. J. POLLARD, "Monte Carlo Methods for Index Computation mod p", *Mathematics of Computation*, **32** (1978), 918-924.
29. JÖRG SAUERBREY AND ANDREAS DIETEL "Resource Requirements for the Application of Addition Chains in Modulo Exponentiation", *Advances in Cryptology – EUROCRYPT '92*, Lecture Notes in Computer Science, **658** (1993), Springer-Verlag, 174-182.
30. R. SCHOOF, "Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p", *Mathematics of Computation*, **44** (1985), 483-494.
31. RICH SCHROEPPEL, HILARIE ORMAN, SEAN O'MALLEY, AND OLIVER SPATSCHECK, "Fast Key Exchange with Elliptic Curve Systems", Univ. of Ariz. Comp. Sci. Tech. Report 95-03 (1995).
32. J. H. SILVERMAN, *The Arithmetic of Elliptic Curves*, Springer Graduate Texts in Mathematics **106** (1992).