Fast Key Exchange with Elliptic Curve Systems

Richard Schroeppel   Hilarie Orman   Sean O'Malley

TR 95 03

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 significantly 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.

March 31, 1995

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

The Diffie-Hellman key exchange algorithm [10] is a very useful method for initiating a 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 [13], 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 [19] to be attackable using a method described in [8]. 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 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 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 [23] and independently by Neal Koblitz [17], 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 $\mathbb{F}_{2^{155}}$. 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 $\mathbb{F}_{2^{155}}$ 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 $\mathbb{F}_{2^{155}}$, especially reciprocals, and a minor improvement in the formula for doubling an elliptic curve point. We suggest how to select a curve with constants that are easily manipulated by software. 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 [22]. See Silverman [29] for a general introduction to elliptic curves; Menezes [21] 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 $\mathbb{F}_{2^{155}}$and satisfying the equation $y^2 + xy = x^3 + ax^2 + b$. a and b are constants, field elements which we will specify later. For a particular choice of a and b, 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 analogue of the Diffie-Hellman key exchange method uses an elliptic curve $E_{a,b}$ defined over a field F, and a point $P = (x_0, y_0)$ which generates the whole addition group of E. E, a, b, F, P, x_0, and y_0 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 $K_A P$ by iterating addition of P using a "double and add" scheme. User A sends the x and y coordinates of the point $K_A P$ to user B. User B selects his own secret multiplier K_B and computes and sends to user A the point $K_B P$. Each user can then compute the point $(K_A K_B)P$. 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 $(K_A K_B)P$ by eavesdropping on this 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 of multiplying a group element by a constant are the Add and Double operations. We also include the Negation operation, since it offers a speed increase. The operations are presented slightly modified from their presentation in [22]. The elliptic curve $E_{a,b}$ is the set of all solutions (x, y) to the equation $y^2 + xy = x^3 + ax^2 + b$. a and b are constants from the field $\mathbb{F}_{2^{155}}$; b must be nonzero. 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 also needed to represent the group identity. We use (0,0) to represent O. (Because $b \neq 0$, $(0, 0)$ is never a solution of our equation.) The Add and Double routines make a special check to see if an input is O.

The Double routine is not just an optimized special case of the Add routine: The Add formula fails when the two input points have equal x coordinates; the formula calls for a division by 0. A check is made for this case; if the y coordinates are also equal, the two input points are equal, and the Double routine is called. If the y coordinates differ, the two points are inverses in the elliptic curve group, so O is returned.

(i) (Rule for Adding two points)
Let $(x_1, y_1) \in E(\mathbb{F}_{2^m})$ and $(x_2, y_2) \in E(\mathbb{F}_{2^m})$ be two points.
If either point is O, return the other point as the sum.
If $x_1 = x_2$ and $y_1 = y_2$, use the Doubling rule.
If $x_1 = x_2$ but $y_1 \neq y_2$, return O as the sum.
If $x_1 \neq x_2$, then $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$, where

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

(ii) (Rule for Doubling a point)
Let $(x, y) \in E(\mathbb{F}_{2^m})$ be a point.

If $x = 0$, then $2(x, y) = O$.
If $x \neq 0$, then $2(x, y) = (x_2, y_2)$, where

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

[22] gives the formula for x_2 as $x_2 = x^2 + b/x^2$. Our formula is equivalent, but saves the multiplication by the constant b.

(iii) (Rule for Negating a point)
Let $(x, y) \in E(\mathbb{F}_{2^m})$ be a point. Then $-1(x, y) = (x, x + y)$.

From these formulas, we can determine the number of field operations required for each kind of elliptic curve operation. We see that an Addition step usually requires eight additions, two multiplications, one squaring, three reductions mod $T(u)$, 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 run time are the multiplications and inversions.

3.2 Choosing the Curve

The constant a in the elliptic curve 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 size of the group depends on the choice of a and b. There is a complicated algorithm due to Schoof [28], with improvements by Atkin, Elkies, Morain, and Couveignes [9] for determining the group order. The order is always close to the number of field elements. 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}) [18]. (If we select $a \neq 0$, the order can have the form $2p$ with p near 2^{154}, giving a small amount of extra security.) Lay and Zimmer [20] give a method for creating a curve with a given order, but we are reluctant to use their scheme, since it produces curves closely related to *rational* curves with an extra structural property called *complex multiplication*. We feel this extra structure might be insecure.

We recommend trying small bs, computing the curve order and checking if the order is of the form $4p$. For curves based on $\mathbb{F}_{2^{155}}$, a few hundred tries may be necessary. One scheme for selecting the bs is to try small values for (x, y) and to compute b from the equation $y^2 + xy = x^3 + b$.

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 [25, 26, 12]. 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 Choosing a Multiplier

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 DH algorithm. We have implemented the straightforward double-and-add approach based on the binary expansion of n. For a random 155-bit multiplier n, computing nP will require 154 doubling steps and an average of 77 addition steps. (The number of addition steps depends on the number of 1 bits in the binary representation of n.) Although the number of doubling steps is roughly fixed, it is possible to reduce considerably the number of addition steps needed. This problem is studied in a large literature on *addition chains* [16, 6, 7, 27]. We mention a few points:

- Menezes [21] discusses the idea of using a low Hamming weight multiplier. If one is content to select from among 2^{128} multipliers, while allowing multipliers up to about 2^{155}, then one can select a multiplier of Hamming weight ≤ 43.

- The operation to negate a point is cheap, so we can use *addition-subtraction chains* instead of simply addition chains. The extra flexibility permits shorter chains.

- If we are using one multiplier for many connections, it pays to invest some effort in finding a good addition-subtraction chain for it. The speedup available from using a good addition-subtraction chain with a random multiplier and novel point is 20-30%.

- Since we are free to select a random multiplier, one approach is to select a random addition-subtraction chain instead. This allows us more selection freedom at each chain step, so we need fewer addition or subtraction steps to generate roughly 2^{155} possible multipliers. Some care must be taken in the selection algorithm, to avoid having some multipliers be overly likely. (This can happen because many steps in the addition-subtraction chains commute.)

- When we know the starting point P ahead of time, we can prepare tables which give a considerable speedup [7]. This is the situation for the first two of the four point multiplications in Diffie-Hellman key exchange.

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 4.4.1).

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.)

We explored the use of the Karatsuba [15] method (see Knuth [16] p. 259 and 536) for multiplication. It turned out to be slightly worse for our particular cases.

Some of the programming tricks used to speed up the multiplication are:

- Use a subroutine that multiplies N words by 1 word, keeping the multiplicand, multiplier, and intermediate product in the registers.
- This subroutine contains a loop that sequences through the bits of the multiplier. Unrolling this loop saves time.
- When the field element size is not an exact multiple of the number of bits in a computer word, there will be a partially used word representing the high order bits of a field element. Our field elements have a 27-bit fragment on both 32- and 64-bit machines. On 64-bit machines, a special subroutine for multiplying by the fragment is worthwhile, because the fragment is much shorter than a word.

4.2.1 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**. Sadly, computer manufacturers have largely ignored the need for an instruction to carry out this operation; nonetheless, it 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. Assume the polynomial to be reduced is

$$P(u) = a_{308}u^{308} + \ldots + a_2u^2 + a_1u + a_0.$$

The reduction mod $u^{155} + u^{62} + 1$ proceeds by reducing each term modulo the trinomial and subtracting it from the result. This can be done very efficiently using shifts and xors. First note that

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

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.)

It would be even better to use a binomial as the modulus. Unfortunately, in fields of characteristic 2, polynomials with an even number of terms are always divisible by $u+1$, so they are always reducible (except for $u+1$ itself).

Some other recommended trinomials are $u^{127}+u^{63}+1$, $u^{140}+u^{65}+1$, $u^{172}+u^{81}+1$, $u^{191}+u^{71}+1$, $u^{223}+u^{91}+1$, and $u^{255}+u^{82}+1$. 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. [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 [21] (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 [21]. [2]

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

> 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 [16] 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 $(\bmod M)$ 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**.

[1] Irreducible trinomials are somewhat sparse: e.g., of the degrees from 100-199, 43 have no irreducible trinomial.

[2] Menezes' inversion scheme for a field element $A(u)$, computes $1/A(u)$ as $A(u)^{2^{155}-2} \bmod T(u)$. This can be done with 10 multiplications and 154 squarings [14].

The algorithm will work whenever $A(u)$ and $M(u)$ are relatively prime, $A(u) \neq 0$, $M(u)$ is odd, and $deg(M) > 0$.

```
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 $B0$, $B1$, ..., $G4$ 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 is useful on register-poor machines.

4.4.1 Dividing out u^k

To find the true reciprocal of $A(u)$, we need to divide $B(u)$ by u^k, working $\mathrm{mod} T(u)$. 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,

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

Operation	SPARC IPC	Alpha
155 bit add	3.4	.22
155 x 155 bit multiply	112.0	7.08
32 x 32	7.9	
64 x 64		1.64
155 bit square	8.1	.81
Modular reduction, 310 bits to 155 bits	3.8	.20
Reciprocal, including divide by u^k $(k = 261)$	279.0	25.72
Double an elliptic curve point	538.0	43.68
Add two elliptic curve points	541.0	44.18
Multiply Ecurve point, 154 doubles and 77 adds	124 msec	9.9 msec
Elliptic curve DH key exchange time (computed)	372 msec	29.8 msec
Ordinary integer DH key exchange time	2674 msec	182 msec

Figure 1: Except for the last two lines, all times are in microseconds.

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 the cache without conflict is crucial to achieving the results reported here.

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.

6 Other Applications

The elliptic curve improvements will be helpful in implementing not only DH key exchange, but also make El Gamal style encryption [11] more attractive. The total effort of signing and checking a signature is less with elliptic curve methods than with RSA.

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 happiest 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 [24] 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: When a reciprocal costs only a few multiplications, then addition-subtraction chains can be used to compute powers; this will allow shorter chains, more than recouping 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 key pairs between hosts, six times as many connections could be made as compared with the modulo p implementations. This fact simplifies the task of designers who might otherwise need to develop secondary key distribution methods and, by reducing the computational cost of the method, it assists in preventing "denial of service" attacks against network hosts which might otherwise become bogged down by repeated requests for key exchanges. 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 J. Menezes for providing us with reference [22].

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] J. Bos and M. Coster, "Addition Chain Heuristics", *Advances in Cryptology – CRYPTO '89*, Lecture Notes in Computer Science, **435** (1990), Springer-Verlag, 400-407.

[7] 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.

[8] D. Coppersmith, A. Odlyzko, and R. Schroeppel, "Discrete Logarithms in $GF[p]$", *Algorithmica*, **1** (1986), 1-15.

[9] Jean-Marc Couveignes and François Morain *Algorithmic Number Theory: First International Symposium*, Lecture Notes in Computer Science, **877** (1994), Springer-Verlag, 43-58.

[10] Whitfield Diffie and M. E. Hellman "New Directions in Cryptography", *IEEE Transactions on Information Theory*, **IT-22**, n. 6, Nov. 1976, pp 644-654

[11] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", *IEEE Transactions on Information Theory*, **31** (1985), 469-472.

[12] 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.

[13] 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

[14] 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.

[15] A. Karatsuba, *Doklady Akademiia Nauk SSSR* **145** (1962), 293-294.

[16] Donald E. Knuth, *Seminumerical Algorithms*, The Art of Computer Programming, **2** Addison Wesley 1969

[17] Neal Koblitz, "Elliptic Curve Cryptosystems", *Mathematics of Computation*, **48** n. 177 (1987), 203-209.

[18] 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.

[19] B. La Macchia and A. Odlyzko, "Computation of Discrete Logarithms in Prime Fields", *Designs, Codes and Cryptography*, **1** (1991), p. 47-62.

[20] 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.

[21] Alfred J. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, 1993.

[22] 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, Oct. 30, 1994.

[23] 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.

[24] Peter L. Montgomery, "Modular Multiplication without Trial Division", *Mathematics of Computation*, **44** (1985), 519-521.

[25] P. van Oorschot and M. Wiener, "Parallel Collision Search with Application to Hash Functions and Discrete Logarithms", presented at the 2nd ACM Conference on Computer and Communications Security, Fairfax, Virginia, November 4, 1994.

[26] J. Pollard, "Monte Carlo Methods for Index Computation mod p", *Mathematics of Computation*, **32** (1978), 918-924.

[27] 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.

[28] R. Schoof, "Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p", *Mathematics of Computation*, **44** (1985), 483-494.

[29] J. H. Silverman, *The Arithmetic of Elliptic Curves*, Springer Graduate Texts in Mathematics **106** (1992).