Efficient Scalar Multiplication on Elliptic Curves

Tommi Meskanen

Department of Mathematics, University of Turku
FIN-20014, Turku, Finland
and
Turku Centre for Computer Science TUCS
e-mail: tommes@utu.fi

Ari Renvall

Department of Mathematics, University of Turku
FIN-20014, Turku, Finland
e-mail: ariren@utu.fi

Paula Steinby

Department of Mathematics, University of Turku
FIN-20014, Turku, Finland
e-mail: pauste@utu.fi

Turku Centre for Computer Science
TUCS Technical Report No 400
March 2001

ISBN 952-12-0806-6
ISSN 1239-1891

Abstract

We present a new general exponentiation algorithm by combining the ideas of some existing algorithms. The algorithm is specially tailored for computing scalar multiplications on some binary elliptic curves.

Keywords: cryptography, elliptic curves

TUCS Research Group
Theory Group: Mathematical Structures in Computer Science

1 On Elliptic Curves

An elliptic curve over the field $\mathbb{F}_{2^m}$ suitable for cryptography is of the form

$$y^2 + xy = x^3 + ax^2 + b,$$

where $a, b \in \mathbb{F}_{2^m}, b \neq 0$. We denote the points on this curve by $E(\mathbb{F}_{2^m})$, supplemented with the point at infinity $\mathcal{O}$. This set of points forms a group w.r.t. addition. The formula for adding two points is $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ is

$$\begin{aligned}
\theta &= \frac{y_1 + y_2}{x_1 + x_2} \\
x_3 &= \theta^2 + \theta + x_1 + x_2 + a \\
y_3 &= \theta(x_1 + x_3) + x_3 + y_1.
\end{aligned}$$

Now $P_3 = (x_3, y_3) = P_1 + P_2$. When $P_1 = P_2$, we call the operation doubling and use due formula

$$\begin{aligned}
\theta &= x_1 + \frac{y_1}{x_1} \\
x_3 &= \theta^2 + \theta + a \\
y_3 &= \theta(x_1 + x_3) + x_3 + y_1.
\end{aligned}$$

Moreover, the inverse of a point is given by $-P_1 = (x_1, y_1 + x_1)$.

There is a clear geometric interpretation of point addition. If we agree that the zero point lies on all lines parallel to y-axis, then it can be shown that all lines cut the curve in three points. The addition is then determined by the rule that the sum of all these three points equals $\mathcal{O}$. Thus, P_1, P_2 and $-(P_1 + P_2)$ are all on the same line; and $-P$ is the other point of the curve that has the same x-coordinate as P.

Parties choosing to use a binary elliptic curve cryptosystem must first agree on certain system parameters. These are the underlying field $\mathbb{F}_{2^m}$, the curve parameters a and b, a base point G lying on the curve, and the degree r of that point. They also have to agree on how to represent field elements. In this paper we always assume a polynomial representation.

The operation of computing $kP = P + \cdots + P$ is called scalar multiplication. This is the basic operation in any cryptographic algorithm using elliptic curves. The assumption making elliptic curve cryptography possible is, that determining k from P and kP is computationally impossible if m is large enough. In other words, computing the discrete logarithm in the elliptic curve group is considered intractable. According to the current knowledge, an elliptic curve cryptosystem over a field $\mathbb{F}_{2^{160}}$ is as secure as an RSA system with a 1024-bit modulus.

For an extensive study on elliptic curves in cryptography we refer to [BSS]. Here it suffices to know that we need an efficient algorithm to calculate scalar multiplication. Apart from efficiency, a further merit for an algorithm would be resistance against so-called implementation attacks. In our setup this implies that the computation time of scalar multiplication should not depend on k.

2 Algorithms for scalar multiplication

In this section we present some efficient algorithms for scalar multiplication. Although they can be used in any group, we adopt our notation and terms from the elliptic curves setup. Thus, our goal is to compute the point kP, and we assume that $\#P = r$, $\lfloor \log_2(r) \rfloor + 1 = n$ and $k = \sum_{i=0}^{n-1} k_i 2^i$ (unless otherwise stated).

It should be noted that in general such algorithms are referred to as exponentiation algorithms. For a good survey on exponentiation algorithms, see [Gor] and [MOV].

2.1 Double-and-Add

The *double-and-add* algorithm is the simplest efficient method for scalar multiplication. The idea is to compute kP as a sum of some points $P_i = 2^i P$. Which P_i's are summed is determined by the multiplier k's binary representation: $kP = (\sum k_i 2^i)P = \sum k_i P_i$.

The amount of memory needed to implement this algorithm is modest; only two points need to be memorized simultaneously:

Algorithm 1.

if $k_0 = 0$ **then** $R \leftarrow \mathcal{O}$ **else** $R \leftarrow P$
for $i = 1$ **to** $n - 1$ **do**
  $P \leftarrow 2P$
  if $k_i = 1$ **then** $R \leftarrow R + P$
return R

Clearly, we need $n - 1$ doublings and $w(k) - 1$ additions, where $w(k)$ is the number of 1's in k's binary representation.

2.2 Halve-and-Add

The *halve-and-add* algorithm is very similar to the double-and-add algorithm. In this case we use halvings instead of doublings. The algorithm is guaranteed to work only if r, the order of the point P, is odd.

If r is odd then it is easy to see that there is a unique point Q in the group generated by P such that $2Q = P$ (namely $Q = (r+1)/2 \cdot P$). We denote this point by $P^{(1)} = 1/2 \cdot P$, and recursively $P^{(i)} = 1/2 \cdot P^{(i-1)} = (1/2)^i \cdot P$.

In double-and-add we computed kP using points $P_i = 2^i P$, in halve-and-add we use points $P^{(i)}$ instead. In order to do this we need to modify the multiplier k. Clearly we have $kP = 2^{n-1}k \cdot P^{(n-1)}$. Thus, if we denote $k' = 2^{n-1}k \pmod r = \sum_{i=0}^{n-1} k'_i 2^i$, then $kP = k'P^{(n-1)}$. The algorithm goes as follows.

Algorithm 2.

$k \leftarrow 2^{n-1}k \pmod r$
if $k_{n-1} = 0$ **then** $R \leftarrow \mathcal{O}$ **else** $R \leftarrow P$
for $i = n-2$ **downto** 0 **do**
  $P \leftarrow \frac{1}{2}P$
  if $k_i = 1$ **then** $R \leftarrow R + P$
return R

In this case we have $n-1$ halvings and $w(k') - 1$ additions. Thus, assuming that the time needed to modify k is negligible, the algorithm is more efficient than double-and-add exactly when halving a point is more efficient than doubling a point. In [Knu] it is shown that in some elliptic curves this is the case. Below we shortly describe the method.

Let P be a point on an elliptic curve $y^2 + xy = x^3 + ax^2 + b$, and assume that r (the order of P) is prime and that $\gcd(n, 4) = 2$, where n is the total number of points on the curve. Then the equation $2X = P$ has exactly two solutions Q and $Q + T$, where T is the 2-torsion point of the curve. Moreover, only one of these (Q) is in the subgroup generated by P. This point is defined as $1/2 \cdot P$. It can be computed using the following method. Let $P = (x, y)$, $Q = (u, v)$ and $\theta_Q = u + v/u$.

1. Solve the equation $z^2 + z + a = x$. This has two solutions (viz. θ_Q and $\theta_Q + 1$); pick one of them and denote it by θ.

2. Compute $w = x(\theta + 1) + y$. If we picked the correct θ (i.e. $\theta = \theta_Q$), then $w = x(\theta + 1) + \theta(u + x) + x + v = \theta u + v = u^2$.

3. Check if the equation $z^2 + z + a = \sqrt{w}$ has any solutions by computing $\mathrm{Tr}(a + \sqrt{w}) = \mathrm{Tr}(a^2 + w)$. If there are no solutions, we picked the wrong θ in step 1. In this case set $\theta = \theta + 1$ and $w = w + x$.

4. Compute $u = \sqrt{w}$.

5. Compute $v = u(u + \theta)$.

Otherwise the method is a straight-forward inversion of the doubling algorithm, but in step 1 we have two choices for θ. The other alternative leads us to the desired point Q, the other to $Q+T$. If $\gcd(n,4)=2$, then we can prove that $Q+T$ does not have any halves. Thus, we can check whether we picked the correct θ using the equation of step 3.

The algorithm requires two field multiplications. Moreover, we need to compute a square root in the field, as well as solve a quadratic equation. The function $x \to \sqrt{x}$ is a linear function in a binary field, so it can be computed by multiplying with the corresponding matrix. The solutions to the equation of step 1 are $L(x+a)+\varepsilon$, where $\varepsilon \in \{0,1\}$ and $L(t) = t^2 + t^{2^3} + \cdots + t^{2^{m-2}}$ (if m is odd). The function L is also a linear function, so it can be computed using its matrix, as well.

We conclude with some remarks. First, in the halving procedure we do not use the y-coordinate at all. Thus, in situations where the y-coordinate is not needed, we may omit step 5 and hence, cope with only one multiplication. Second, we assumed that $\gcd(n,4) = 2$. This may seem too restrictive; however all binary standard curves for cryptography (see ANSI X9.62 or WAP-WTLS, for example) satisfy this condition. Third, the two linear functions we needed can be calculated more efficiently if a normal basis instead of a polynomial basis is used. This follows since both squaring and square-rooting is then performed via a simple rotation of the coordinates.

2.3 Fixed-Base Windowing

In the previous algorithms we took advantage of the binary representation of k. In the *fixed base windowing* (FBW for short) we use k's representation in some base 2^w. Denote $l = \lceil \frac{n}{w} \rceil$ and (contrary to the previous algorithms) $k = \sum_{i=0}^{l-1} k_i 2^{iw}$ $(0 \le k_i < 2^w)$. We also denote $P_i = 2^{iw}P$ $(i = 0,1,\ldots,l-1)$; hence $kP = \sum_{i=0}^{l-1} k_i P_i$.

Algorithm 3.

$R \leftarrow 0$, $B \leftarrow 0$
$T[0] \leftarrow P$
for $i = 1$ **to** $l-1$ **do**
    $T[i] \leftarrow 2^w T[i-1]$
for $i = 2^w - 1$ **downto** 1
    for $j = 0$ **to** $l-1$ **do**
        if $k_j = i$ **then** $B \leftarrow B + T[j]$
    $R \leftarrow R + B$
return R

First in the algorithm we generate the precomputation table T, where $T[i] = P_i$, and then we start to calculate the result to R. Clearly each $T[i]$ is included in the increment B in k_i loops, so in the end $R = \sum k_i P_i = kP$.

In the precomputation phase we need $(l-1)w$ doublings, and in the actual computation $l + 2^w - 3$ additions. Thus, a suitable choice for w

somewhat reduces the number of necessary operations compared with the previous algorithms. The cost is the increased need of memory, as now a total of $l + 2$ points needs to be simultaneously in memory.

2.4 Signed-Digit Representation

In the double-and-add algorithm the number of additions needed depends on the number of 1's in k's binary representation. One way to make the method more efficient is to modify k in such a way that its representation has more 0's. This can be done using a so-called *signed-digit representation* for k, where also -1's are used. This is best described using an example: $7 = 2^2 + 2^1 + 2^0 = (111)_2 = (100\bar{1})_s = 2^3 - 2^0$ (where $\bar{1} = -1$). It is easily seen that for any integer we can find a signed-digit representation where no adjacent digits are non-zero. (As a consequence the length of the representation might grow by one.)

It is pretty obvious how to use a signed-digit representation in scalar multiplication. In the following we assume that k is already given in this form: $k = \sum_{i=0}^{n-1} k_i 2^i$ ($k_i \in \{-1, 0, 1\}$).

Algorithm 4.

$R \leftarrow \mathcal{O}$
if $k_0 = 1$ **then** $R \leftarrow P$
if $k_0 = -1$ **then** $R \leftarrow -P$
for $i = 1$ **to** $n - 1$ **do**
    $P \leftarrow 2P$
    if $k_i = 1$ **then** $R \leftarrow R + P$
    if $k_i = -1$ **then** $R \leftarrow R - P$
return R

As in the double-and-add algorithm, we need $n-1$ doublings and $w(k)-1$ additions. However, in this case we have $w(k) \leq \lceil (n+1)/2 \rceil$. Moreover, we need to compute inverses $-P$. (In the elliptic curve case the cost of this is negligible compared with the cost of one addition).

3 An efficient combined algorithm

In this section we combine the ideas of the algorithms presented in the previous section. The algorithm is specially tailored to perform scalar multiplication on such elliptic curves over a binary field, where halving a point is easy.

In the following we denote $l = \lceil \frac{n}{w} \rceil$ and $P^{(i)} = (1/2)^{iw} \cdot P$ for $i = 0, 1, \ldots, l-1$.

The main ideas of our algorithm are listed below.

- We exploit the equation $kP = k'P^{(l-1)}$, where $k' = 2^{(l-1)w}k \pmod r$ using the idea of the halve-and-add algorithm.

- The point $k'P^{(l-1)}$ is computed using the FBW algorithm. To this end we need the precomputation table T for $P^{(l-1)}$, consisting of the points $P, P^{(1)}, \ldots, P^{(l-1)}$.

- The table T is computed via repeatedly halving P. As we already noted, this suits well to elliptic curve case.

- The signed-digit techniques is used to modify k' in such a way that each $-2^{w-1}+1 \leq k'_i \leq 2^{w-1}$. If we have $k'_i < 0$ then $-P^{(i)}$ (instead of $P^{(i)}$) is correspondingly incremented to B.

The description of the algorithm follows:

Algorithm 5.

$k = \sum k_i 2^{iw} \leftarrow 2^{(l-1)w}k \pmod r$
for $i = 0$ **to** $l-1$ **do**
    if $k_i > 2^{w-1}$ **then**
        $k_i \leftarrow k_i - 2^w$
        $k_{i+1} \leftarrow k_{i+1} + 1$
$T[l-1] \leftarrow P$
for $i = l-2$ **downto** 0 **do**
    $T[i] \leftarrow (1/2)^w T[i+1]$
$R \leftarrow 0,\ B \leftarrow 0$
for $i = 2^{w-1}$ **downto** 1 **do**
    for $j = 0$ **to** $l-1$ **do**
        if $k_j = i$ **then** $B \leftarrow B + T[j]$
        if $k_j = -i$ **then** $B \leftarrow B - T[j]$
    $R \leftarrow R + B$
return R

When considering the complexity of this algorithm, we ignore the cost of modifying k. This is reasonable, as (at least in the elliptic curve case) the time required for those operations is negligible compared to others.

The precomputation phase is analogous to the basic FBW algorithm: we need $(l-1)w$ operations, in this case halvings. The time needed for the actual computation is reduced, as now the number of iterations in the outer loop (determined by parameter i) is halved (thanks to the signed-digit technique). Now we need $l + 2^{w-1} - 2$ point additions (or subtractions). Again, the optimal choice for w depends on the order of P.

To perform the algorithm we need memory to store $l+2$ points (l for T, 1 for R and S). If we are short of memory, this can be reduced with a slight modification. Clearly each $T[i]$ is used exactly once (when incrementing it to B). Instead of saving T we could compute and save the values for the increment B in each loop. In this method only $2^{w-1}+3$ points need to be stored in the memory.

One more thing should be noted. When calculating the necessary amount of memory we have assumed that doubling or halving a point can be done without extra memory. This is not always the case. In the efficient point halving method with binary elliptic curves we need to compute two linear functions. Probably the most efficient way to do this is to use matrices stored in the memory. The size of one matrix corresponds roughly the size of n points.

4 Performance

In previous sections the complexities of the presented algorithms were computed in terms of point operations. We cannot get more precise unless the domain is specified. Let us now consider more closely the application we have in mind: elliptic curves over a binary field, where a polynomial representation of field elements is chosen.

To obtain a more accurate comparison between our new algorithm and the previous ones, we measure the times needed to perform the point operations. Naturally, these depend on the implementation of the basic arithmetic operations of the base field. Below we list these operations and briefly explain our implementation.

field addition The time needed for adding two field elements is not significant, as it is performed by simply xoring two bit vectors.

field multiplication To multiply two field elements together we use Karatsuba multiplication described in [MOV].

field squaring The cost for squaring field elements is smaller than for ordinary multiplication. This is because we can simply square all the terms of an element separately and then add them up.

field inversion This is the most time-consuming operation. We use the almost inverse algorithm of [SOOS]. It is a faster version of the extended Euclidean algorithm.

linear mappings When halving points on the elliptic curve we have to compute two linear functions ($\sqrt{\ }$, L). We implemented these by storing their matrices in memory. The cost is then approximately the same as for one multiplication.

In the table below we give the computing times for the relevant operations. To this end we need to fix the size of the field. We choose a field of (from cryptographic point of view) realistic size: $\mathbb{F}_{2^{163}}$. The time unit is 1000 clock cycles using a Pentium III processor.

Operation	time
field multiplication	19
field squaring	9
field inversion	62
linear mapping	19

Let us next consider the point operations we use.

point addition Adding two points on an elliptic curve requires 1 squaring, 2 multiplications and 1 inversion in the base field.

point doubling The cost for doubling a point is exactly the same as for adding two points.

point halving The efficient point halving algorithm costs 1 field multiplication and two linear mappings. One additional multiplication is required if the y-coordinate of the resulting point is needed.

The times needed for these operations will then be as follows.

Operation	time
point addition	110
point doubling	110
point halving	60

Let P be a point of prime order r, where $2^{161} < r < 2^{162}$. Thus, our task is to compute the point kP, where k is a 162-bit integer.

In the next table we give the computation times for kP using each of the presented algorithms. We have included the number of doublings, ordinary additions and halvings in average (on left) and in the worst case (on right). The computation times are given in millions of clock cycles. In the fixed-base windowing and in our new combined algorithm we have chosen $w = 4$ (and hence $l = 41$).

Algorithm	$2P$		$P+Q$		$\frac{1}{2}P$		time	
double-and-add	161	161	80	161	-	-	27	35
halve-and-add	-	-	80	161	161	161	18	27
signed-digit repr.	162	162	54	80	-	-	24	27
fixed-base windowing	160	160	52	54	-	-	23	23
combined algorithm	-	-	45	47	160	160	15	15

We notice that in the worst case our new combined algorithm is approximately 35% faster than the basic fixed-base windowing algorithm, and the difference is even bigger w.r.t. the other algorithms. On the other hand, in the average case the simple halve-and-add algorithm is almost as quick as our new algorithm. However, in some cryptographic setups we prefer constant time algorithms (to avoid so-called timing attacks); in those cases halve-and-add must be ruled out.

5 Conclusions

We have presented a new general exponentiation algorithm by combining the ideas of some existing algorithms. It suits specially well to compute scalar multiplications in the group of points on some elliptic curves over binary fields. Apart from being efficient, it is also a constant-time algorithm, which is a merit in a cryptographic setup.

References

[BSS] I. Blake, G. Seroussi and N. Smart, Elliptic Curves in Cryptography, Cambridge University Press, Cambridge, 1999.

[Gor] M. D. Gordon, A Survey of Fast Exponentiation Methods, Journal of Algorithms 27 (1998), 129-146.

[Knu] Erik Woodward Knudsen, Elliptic Scalar Multiplication Using Point Halving, Advances in Cryptology - Asiacrypt'99, LNCS 1716, Springer-Verlag, 1999, pp.135-149.

[MOV] A. Menezes, P. van Oorschot and S. Vanstone, Handbook of Applied Cryptography, CRC Press, Boca Raton, Florida, 1996.

[SOOS] R. Schroeppel, H. Orman S. O'Malley and O. Spatscheck, Fast Key Exhange with Elliptic Curve Systems, Advances in Cryptology - CRYPTO 95, LNCS 963, Springer-Verlag, 1995, pp.43-56.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**