Contents

1 Introduction **2**
1.1 Diffie-Hellman Key Exchange . . . 2
1.2 A Candidate One-Way Function . . . 3

2 The Discrete Logarithm Problem **4**
2.1 Bounds on Generic Algorithms . . . 4
2.2 Implications for the DLP as a One-Way Function . . . 4
2.3 A Generic 'Square Root' Algorithm . . . 5

3 Addition Chains **5**
3.1 Fast Exponentiation . . . 6
3.2 Faster Exponentiation . . . 7
3.2.1 m-ary algorithms . . . 7
3.2.2 Windowing techniques . . . 8
3.3 Bounds . . . 8

4 Computing with Elliptic Curves over Finite Fields **8**
4.1 Affine description of the group law . . . 9
4.2 Projective description of the group law . . . 9
4.3 Side Channel Attacks . . . 10
4.4 Alternative coordinate systems . . . 10
4.4.1 Edwards Form . . . 11

5 References and further reading **12**

Computational aspects of elliptic curve cryptography.

Graeme Taylor

April 18, 2008

1 Introduction

Cryptography is the study of secret sharing via coded messages. Since at least Roman times, cryptosystems have been employed that scramble a *plaintext* message into safely-transmittable *ciphertext*. Of course, to unscramble the message at the other end, one needed to know the secret of the system; the rule itself, or in modern implementations a crucial parameter known as the *secret key*. Historically, one might have been able to share this secret face-to-face, and keep it secret within a government or military organisation. But modern needs such as online shopping require the exchange of messages between previously unrelated participants- how then to overcome this first hurdle?

Towards the end of the 20th century, the notion of *public key cryptography* was developed to resolve this problem. The *Diffie-Hellman key exchange protocol* describes a means for two agents to negotiate a shared secret, without any third party who intercepts their negotiations being able to deduce the secret themselves.

To do so, public key cryptography depends upon the idea of a *one-way function*. This is a function with the property that it is exponentially harder to invert than to compute- even with knowledge of some of the inputs.

1.1 Diffie-Hellman Key Exchange

In the context of Diffie-Hellman key exchange, we will consider a one-way function $f : X_1 \times X_2 \to X_2$ with the additional property that

$$f(x, f(y, z)) = f(y, f(x, z)) \tag{1}$$

A shared secret S between Alice and Bob can be established as follows:

- Publically agree upon a fixed element $M \in X_2$.
- Alice selects $a \in X_1$ as her private key, and generates public key $A = f(a, M) \in X_2$.

- Bob selects $b \in X_1$ as his private key, and generates public key $B = f(b, M) \in X_2$.
- These public keys can be safely exchanged in public as a, b cannot reasonably be recovered from them.
- Alice computes $S = f(a, B)$.
- Bob computes $f(b, A) = f(b, f(a, M)) = f(a, f(b, M)) = f(a, B) = S$.
- Thus Alice and Bob now know S, but an eavesdropper cannot without inverting f to determine a private key.

1.2 A Candidate One-Way Function

Crucially, no one has proven the existence of a one-way function!

A candidate arises from finite groups. We will work additively throughout[1]. For such a group $G, \oplus$ of order N, define the scalar multiple $[t]g$ of $g \in G$ by

$$\underbrace{g \oplus g \oplus \cdots \oplus g}_{t \text{ copies}}$$

So we can consider the map

$$f : \mathbb{N} \times G \to G$$

$$f(n, g) = [n]g$$

Since G has order N, we can restrict our attention to $\mathbb{Z}/N\mathbb{Z}$ instead of $\mathbb{N}$.

Notice this satisfies (1) since

$$f(x, f(y, z)) = f(x, ([y]z)) = \underbrace{[y]z \oplus [y]z \oplus \cdots \oplus [y]z}_{x \text{ copies}} = \underbrace{z \oplus z \oplus \cdots \oplus z}_{xy \text{ copies}} = \underbrace{[x]z \oplus [x]z \oplus \cdots \oplus [x]z}_{y \text{ copies}} = f(y, f(x, z))$$

The rest of this talk will consider the implications of the one-way requirement:

- Given $h = [t]g$ and g, how hard is it to recover t? (the *discrete logarithm problem*)
- How easily can we compute $[t]g$ from t, g? (addition chains)
- What conditions do we have on G, and how might we compute with such a group?
- (briefly) What about non-mathematical attacks?

[1] For multiplicative groups, replace addition by multiplication; scalar multiplication becomes scalar exponentiation.

2 The Discrete Logarithm Problem

2.1 Bounds on Generic Algorithms

We consider *generic algorithms*- ones which see the group as a 'black box', and thus may only perform the *group operations* of addition ($g, h \mapsto g \oplus h$), inversion ($g \mapsto -g$), and equality testing ($g, h \mapsto g = h \in \{True, False\}$). Our question then becomes, *how many such group operations are necessary to solve the DLP* $h = [t]g$*?*

For a fixed g generating cyclic G of order n. Suppose that we represent each element of G uniquely as some binary string $s \in S$. Clearly then S has at least n elements; let p be the largest prime dividing n. There is an isomorphism

$$e : \mathbb{Z}/n\mathbb{Z} \to G$$
$$e(k) = [k]g$$

and thus a map

$$\sigma : \mathbb{Z}/n\mathbb{Z} \to S$$

such that the DLP becomes: *given* $\sigma(1), \sigma(t)$*, find* t.

In such a setting, Shoup (97, see [HB] Thm 19.2) presents the desired result- if σ is chosen randomly, and $\mathcal{A}$ is an algorithm that reads in $\sigma(1), \sigma(t)$, performs m group operations and then returns an answer $v \in \mathbb{Z}/n\mathbb{Z}$ then the probability that $t = v$ is $O(m^2/p)$.

That is, to have a non-negligible probability of success, we must perform on the order of $\sqrt{p}$ group operations. For a difficult DLP, we are therefore interested in large prime groups (or at least, with large prime factors).

2.2 Implications for the DLP as a One-Way Function

Doesn't this contradict the previous claim that no one-way functions are known? No, since any implementation must specify a group G, and additional properties of that group may be exploitable by a non-generic algorithm. The randomness of σ is important too- if there is an obvious (i.e., easily computable) isomorphism between G and $\mathbb{Z}/p\mathbb{Z}$ for a prime p then the DLP is easy!

Example 1 (DLP in $\mathbb{Z}/p\mathbb{Z}$)**.** Suppose $G = \mathbb{Z}/p\mathbb{Z}$, $g = \lceil a \rceil$ i.e., the equivalence class of some a, and $h = \lceil b \rceil$ satisfies $h = [t]g$. Then

$$\lceil b \rceil = h = [t]g = [t]\lceil a \rceil = [\lceil t \rceil]\lceil a \rceil = \lceil ta \rceil$$

so $b \equiv ta$ modulo p. Thus $t \equiv ba^{-1}$. But we can easily recover $a^{-1} \in \mathbb{Z}/p\mathbb{Z}$ by the extended Euclidean algorithm (as a, p are coprime there are u, v with $1 = au + pv$, mod p u is therefore the inverse of a) - this is a single group operation - then a scalar multiplication by b (see next section) efficiently recovers t.

Essentially, we exploited the fact that t can be thought of as an element of G. Cyclic groups of rational points of an ellipitc curve are of interest because such a trick does not seem possible.

2.3 A Generic 'Square Root' Algorithm

Working purely generically, we can attain the performance implied by Shoup's result- that is, compute the DLP after on the order of $\sqrt{p}$ group operations. Such algorithms are described as *square root algorithms* and we present one here, the *Baby-Step, Giant-Step algorithm* (BSGS). We will assume the availability of *hash tables*: this data structure lists elements, and can be queried for the existence of an entry in constant time regardless of its size[2]. This eliminates the cost of equality testing, and thus we assess performance in terms of additions and inversions (although none of those are required). Suppose for a motivating example that we know $[t]g = h$ for some $t \in 0 \dots 99$.

Example 2 (Naive testing)**.** The obvious approach is to compute successive group elements $g, [2]g, [3]g, \dots$ and test each one for equality to h, stopping once a match is made. At worst, this will require 100 additions, and on average around 50.

Example 3 (BSGS testing)**.** The BSGS approach is to split n and test multiple cases at once. Since n is a two digit number, we have $n = 10a + b$ for some a, b from 0 to 9. Hence $[10a + b]g = h$, i.e.,

$$[10a]g = h - [b]g$$

We first therefore list the 10 possible *baby steps* for the RHS: $h, h - g, h - [2]g$ etc. in the hash table. This requires 10 additions, after an inversion determines $-g$. Then, we start to build *giant steps* for the LHS: $[10]g, [20]g, \dots$ until a match is obtained. At worst, there will be 20 additions in total; on average, 15.

In general, given a bound p on t (such as the group order) and fixing some s, we can write $n = as+b$, $0 \leq b < s$ (by the remainder theorem), commiting to s baby steps and at most p/s giant steps. For instance, we can see the naive search as the special case $s = 1$ (only 'giant' steps). The larger s the more powerful each giant step, but the more baby steps one must precompute. The optimal choice is thus the closest integer to $\sqrt{p}$ to give a worst case of $p/\sqrt{p} + \sqrt{p} = 2\sqrt{p}$ steps/additions, that is, on the order of $\sqrt{p}$ as desired.

3 Addition Chains

Multiplication is just repeated addition, but unless you're at primary school you're unlikely to compute 16×7 as $7 + 7 + 7 + \dots + 7$. Instead, you might do something like

$$10 \times 7 = 70$$

$$5 \times 7 = 35$$

Hence

$$16 \times 7 = 70 + 35 + 7 = 112$$

[2]For our purposes at least, this is valid: a far more detailed examination of hashing would go here if I was giving a lecture course rather than a talk!

In base 10, multiplication by powers of ten is very easy, and thus we can break a multiplication down into simpler steps. In a generic group, we do not have this luxury- or do we we?

3.1 Fast Exponentiation

If you can add, then you can double- just as we exploit easy multiplication by 10 in base 10, so we can try to make use of doubling by thinking in base 2.

Example 4 (Computing 16)**.** For a power of 2, we only need to double. For instance, to compute $[16]g$, instead of performing 15 additions, we need only perform 4:

$$[16]g = [8]g \oplus [8]g$$

$$[8]g = [4]g \oplus [4]g$$

$$[4]g = [2]g \oplus [2]g$$

$$[2]g = g \oplus g$$

In general, we will be able to compute $[n]g$ for n a power of 2 in $lg(n) := log_2(n)$ steps.

Definition 5. We describe a list $\alpha_1 \dots \alpha_k$ as an *addition chain* for n if $\alpha_1 = 1$, $\alpha_k = n$ and for each α in the chain beyond the first, there are α_i, α_j in the chain such that $\alpha = \alpha_i + \alpha_j$ (n.b., $i = j$ is valid). The *length* of the chain is $k - 1$.

Example 6 (Addition chains for 16)**.** By the previous example, an addition chain for 16 is $1, 2, 4, 8, 16$ of length 4. Notice that $1, 2, 4, 6, 8, 16$ is also a chain for 16, of length 5, although the 6 is redundant.

We will be interested in producing chains as short as possible, and minimal in the sense that if any term is removed, it fails to be an addition chain (the chain 1,2,4,6,8,16 above is thus not minimal).

Since we can form any power of 2, and the binary representation of any n expresses it as a sum of powers of 2, we can certainly construct an addition chain for n by computing all necessary powers of 2, and then adding as appropriate.

Example 7 (Computing $17g$)**.** Since $17 = 2^4 + 2^0$, we could compute each of $[2]g, [4]g, [8]g, [16]g$ then sum as appropriate ($[16]g \oplus g$). The addition chain is then 1,2,4,8,16,17 of length 5.

However, we can avoid the need to store all $lg(n)$ powers of 2 by instead reading the binary representation such that each step in the chain needs only the previous one and g. Suppose that we have computed $[m]g$ where m has binary representation $(a_0 \dots a_k)_2$. Then the number with with binary representation $(a_0 \dots a_k 0)_2$ is $2m$, which we can obtain by a double of $[m]g$, whilst the number with binary representation $(a_0 \dots a_k 1)_2$ is $2m + 1$, which we reach by a double of $[m]g$ then an addition of g. In this way, we can construct $[(a_0 \dots a_l)_2]g$ for any $n = (a_0 \dots a_l)_2$, reading left to right. This gives us the *fast exponentiation algorithm.*

Example 8 (Fast Exponentiation Algorithm)**.** INPUT: $n = (a_0 \dots a_l)_2$, g; OUTPUT: $[n]g$

- Set $x = g$.
- For i from 1 to l:
  - Set $x = x \oplus x$.
  - If $a_i = 1$, set $x = x \oplus g$
- Return x

Notice that there will be $lg(n)$ steps, and each step requires either 1 or 2 group operations. Defining the *Hamming weight* of n, $v(n)$, to be the number of 1's in its binary expansion, we will obtain a chain of length $lg(n) + v(n) - 1$. At best, for a power of 2, we thus require $lg(n)$ additions; for a random string (50/50 chance of each digit being 0 or 1) $(3/2)lg(n)$. Worst case performance arises for a number with binary representation entirely 1's:

Example 9 (Fast Exponentiation of 15)**.** Using the algorithm above, we require 6 additions, via the chain 1,2,3,6,7,14,15.

Fast Exponentiation is already good enough for our purposes of constructing a one-way function from scalar multiplication and the DLP. If we have a group G_1 of prime order P_1, then we need never compute anything greater than $[P_1]g$, which takes $O(lg(P_1))$ group operations. If G_2 is a group of prime order $P_2 \approx P_1^2$, then the runtime doubles, since $lg(N^2) = 2lg(N)$. But then the generic DLP changes from taking $O(\sqrt{P_1})$ to $O(\sqrt{P_1^2}) = O(P_1)$ group operations. That is, the runtime of the inverse grows exponentially with an increase in the forward direction (doubling forward squares inverse). So we can choose an appropriate P_1 such that the inversion is infeasible by generic methods whilst the forward is still easily computable.

3.2 Faster Exponentiation

We observed that fast exponentiation performs poorly for high Hamming weights. The next example shows that it is not optimal:

Example 10 (Addition chain for 15)**.** A length 5 addition chain for 15 is possible, using terms 1,2,3,5,10,15.

3.2.1 m-ary algorithms

For such a small number, this chain could be recovered by brute force. However, it corresponds to a tertiary (base 3) expansion of 15. In general, we can work with a base m representation, but instead of double and double-and-add, we will need multiply-by-m, multiply-by-m-and-add-1, multiply-by-m-and-add-2 etc. through to multiply-by-m-and-add-$(m-1)$ to cover the possible digits. The multiplication by m is itself an addition chain problem, and thus powers of 2 are a popular choice. Less steps are required ($log_m(n)$) but each step is more expensive, and there is a precomputation and storage overhead, so an appropriate choice of m depends on typical values for n.

Example 11 (m-ary Exponentiation Algorithm)**.** INPUT: $n = (a_0 \dots a_l)_m$, g; OUTPUT: $[n]g$

- Precompute and store $[2]g, [3]g, \dots, [m-1]g$
- Set $x = [a_0]g$.
- For i from 1 to l:
  - Set $x = [m]$.
  - If $a_i \neq 0$, set $x = x \oplus [a_i]g$
- Return x

3.2.2 Windowing techniques

Instead of working with (potentially awkward) base-m representations, one can instead tackle the binary case by looking at more than 1 bit at a time. These are known as windowing techniques. For instance, if [k]g has been computed and the next two bits are both a 1, then the fast exponentiation route will proceed as $[2k]g, [2k+1]g, [4k+2]g, [4k+3]g$, taking 4 additions. A better approach is to precompute [3]g, then form the chain $[2k]g, [4k]g, [4k+3]g$ in three additions. For k-bit windows, this will require k doublings and at most one addition per window (instead of k doublings and up to k additions for fast exponentiation); but we must precompute up to $[k-1]g$ as in the m-ary algorithm.

Instead of looking at each successive block of k bits, one can adopt a sliding windows approach, proceeding by doubling through runs of zeros until a 1 is encountered, at which point the number of nonzero bits (up to window size) can be found to determine the appropriate addition and then revert to doubling.

3.3 Bounds

Erdős proved a lower bound for the length of an addition chain of $lg(n) + lg(v(n))$.

No algorithm is known for the generation of an optimal length addition chain for an arbitrary n. Further, the related question of finding an optimal chain containing $n_1, \dots, n_k$ for some k (the *addition sequence problem*) has been shown to be NP-hard. There has been much interesting work in this area; for instance, the use of genetic algorithms to "breed" addition chains. Time spent optimising the addition chain must of course lead to greater savings in time saved performing addition to be worthwhile.

4 Computing with Elliptic Curves over Finite Fields

So far, we've assumed that we have a finite cyclic group with prime order (or at least, a large prime factor) such that the isomorphism to $\mathbb{Z}/p\mathbb{Z}$ is sufficiently random and thus there is no way

to interpret scalars as elements of the group. Groups of rational points of an elliptic curve over a finite field (the full group, or a cyclic subgroup) are believed to be suitable, with attacks on the DLP only known for special cases. Thus we are interested in explicit computation with such points.

Under appropriate conditions, we can think of an elliptic curve as the affine piece $y^2 = x^3 + a_4x + a_6$, and a "point at infinity" ∞. This allows for a geometric description of the group law: any straight line will intersect the curve at three points (counting multiplicity as appropriate, and considering a vertical line to have third intersection at ∞). Then, to determine $P \oplus Q$ one first finds $P * Q$ the third point of intersection of the cubic with the line joining P and Q (use tangent at P if $P = Q$); then takes $P \oplus Q$ to be the third intersection with the line joining $P * Q$ and ∞.

(Sorry, no picture!)

However, such a description is not practical for a computer. It might also seem specific to $\mathbb{R}$-over $\mathbb{F}_5$, for instance, how does one even draw the curve?

The way out is to describe the group law algebraically, verify that it does indeed give a group, and that the equations hold over any field K.

4.1 Affine description of the group law

For points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, the point $P \oplus Q = (x_3, y_3)$ is given by

$$x_3 = \lambda^2 - x_1 - x_2$$

$$y_3 = -\lambda x_3 - \nu$$

Where

$$\lambda = \begin{cases} (y_2 - y_1)/(x_2 - x_1) & \text{if } x_1 \neq x_2 \\ (3x_1^2 + a_4)/2y_1 & \text{if } x_1 = x_2 \end{cases}$$

$$\nu = \begin{cases} (y_1x_2 - y_2x_1)/(x_2 - x_1) & \text{if } x_1 \neq x_2 \\ (-x_1^3 + a_4x_1 + 2a_6)/2y_1 & \text{if } x_1 = x_2 \end{cases}$$

Notice that we still require a symbolic representation of ∞, and have to catch the cases where either or both is this point: or $P = -Q$ and thus the answer should be ∞... these are all problematic from a data structure point of view. Over finite fields, the inversions in the formulae above are also undesirable, taking longer to compute than multiplications and squarings.

4.2 Projective description of the group law

We resolve these issues by moving to a projective description of the curve:

$$Y^2Z = X^3 + a_4XZ^2 + a_6Z^3$$

with any point being a triple $(X : Y : Z)$ (X, Y, Z not all zero) modulo multiplication by nonzero scalars. Affine points (x, y) are identified with $(x : y : 1)$, $\infty = (0 : 1 : 0)$, and

$-(X_1 : Y_1 : Z_1) = (X_1 : -Y_1 : Z_1)$. Now we can simply work with a triple of points, and since $(X : Y : Z) = (kX : kY : kZ)$ we can clear denominators and thus obtain inversion-free group law formulae, although one must still check for cases $P = \pm Q$ or $P, Q = \infty$.

To double a point $(X_1 : Y_1 : Z_1)$:

$$A = a_4 Z_1^2 + 3X_1^2 \, , \, B = Y_1 Z_1 \, , \, C = X_1 Y_1 B \, , \, D + A^2 - 8C$$

Then

$$X_3 = 2BD \, , \, Y_3 = A(4C - D) - 8Y_1^2 B^2 \, , \, Z_3 = 8B^3$$

To add points $(X_1 : Y_1 : Z_1) \neq \pm(X_2 : Y_2 : Z_2)$:

$$A = Y_2 Z_1 - Y_1 Z_2 \, , \, B = X_2 Z_1 - X_1 Z_2 \, , \, C = A^2 Z_1 Z_2 - B^3 - 2B^2 X_1 Z_2$$

Then

$$X_3 = BC \, , \, Y_3 = A(B^2 X_1 Z_2 - C) - B^3 Y_1 Z_2 \, , \, Z_3 = B^3 Z_1 Z_2$$

These formulae have been somewhat optimised, with a cost of 12M 2S for the general add and 7M 5S for the double. That a double is cheaper than an add further motivates the elimination of additions in the previous section. Notice, however, that a doubling *cannot* be performed using the general addition formulae - an answer of $(0 : 0 : 0)$ is obtained - and in certain contexts this is undesirable.

4.3 Side Channel Attacks

Suppose we have a device such a smart card that contains a secret key k. Under the one way assumption, publishing the value of $[k]g$ does not compromise this key. But what if one could see the "working out" done by the device? If the components for doubling and addition are different, then analysis of power consumption, heat output or other physical attributes could disclose which series of calculations is being performed. As we can read off the addition chain from the binary digits of k, so we can use the observed sequence to recover those digits and hence k. This is known as a *side channel attack*.

Resistance to SCA requires obscuring the nature of the calculation being performed: a particularly simple, but very costly way to do this is to adopt a "double and always add" strategy of performing an unneeded double for every add, and an add for every double. More sophisticated obfuscation schemes are possible, but these evidently require some wasteful decoy calculations. Preferable is to have a *unified group law*, such that a single formula can handle $P \oplus Q$ even in the case $P = Q$, so that doubling behaves no different at chip level to addition.

4.4 Alternative coordinate systems

At the other extreme, various coordinate systems have been developed to try and cut down the costs of addition and/or doubling, for contexts where SCA is not a risk but performance is crucial.

These may be optimised for particular tasks, such as algorithms where readdition of a previously added point is common, or doubling is particularly common.

Recently (April 07) a new normal form for elliptic curves, the Edwards form, was introduced. This has generated considerable interest, since it offers both a unified group law for SCA resistance which is already faster than all general addition formulae known, and where SCA is not a concern has a doubling formula which is the second fastest known[3].

4.4.1 Edwards Form

Edwards demonstrated ([Ed]) that every elliptic curve over some K can be written in the form

$$x^2 + y^2 = a^2(1 + x^2y^2) \;\; a^5 \neq a$$

although this may require lifting to an extension field of K. Bernstein and Lange [BeLa] give a more general form which captures more curves over their original field of definition:

$$x^2 + y^2 = c^2(1 + dx^2y^2) \;\; c, d \neq 0 \;\; dc^4 \neq 1$$

and further, any such curve can be transformed into one with $c = 1$.

Notice that the neutral element in this form is an affine point, $(0, c)$, and the inverse of (x, y) is $(-x, y)$. The unified group law is

$$(x_1, y_1) \oplus (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{c(1 + dx_1x_2y_1y_2)}, \frac{y_1y_2 - x_1x_2}{c(1 - dx_1x_2y_1y_2)} \right)$$

Working projectively to avoid inversion, homogenize to

$$(X^2 + Y^2)Z^2 = c^2(Z^4 + dX^2Y^2)$$

Then an efficient implementation of the group law costing 10M 1S and a scalar multiplication by each of c and d is given by

$$A = Z_1Z_2 \;\; B = A^2 \;\; C = X_1X_2 \;\; D = Y_1Y_2$$

$$E = (X_1 + Y_1)(X_2 + Y_2) - C - D \;\; F = dCD$$

$$X_3 = AE(B - F) \;\; Y_3 = A(D - C)(B + F) \;\; Z_3 = C(B - F)(B + F)$$

A non-unified doubling is much faster. The affine formula is

$$[2](x, y) = \left(\frac{2cxy}{x^2 + y^2}, \frac{c(y^2 - x^2)}{2c^2 - (x^2 + y^2)} \right)$$

working projectively, one can achieve such a doubling in 3M 4S plus the cost of three multiplications by c (which can be transformed to 1):

$$B = (X + Y)^2 \; C = X^2 \; D = Y^2 \; E = C + D \; H = (cZ)^2 \; J = E - 2H$$

$$X_3 = c(B - E)J \;\; Y_3 = cE(C - D) \;\; Z_3 = EJ$$

[3] Based on costings in [BeLa].

5 References and further reading

I attended two talks by Dan Bernstein and Tanja Lange on the Edwards form, at ECC07 and SAGE Days 6. They maintain a large amount of information on this at *http://cr.yp.to/newelliptic.html*, including the papers [BeLa] and [Ed] listed below.

References

[BeLa] *Faster addition and doubling on elliptic curves* Daniel J. Bernstein and Tanja Lange.

[Ed] *A Normal Form for Elliptic Curves* Harold M. Edwards Bulletin of the AMS 44 (2007)

[HB] *Handbook of Elliptic and Hyperelliptic Curve Cryptography* Cohen, Frey et al. Chapman and Hall 2006

[Go] *A Survey of Fast Exponentiation Methods* Daniel M. Gordon Journal of Algorithms 27 (1998)