On the Performance of Signature Schemes based on Elliptic Curves

Erik De Win[1⋆], Serge Mister[2], Bart Preneel[1⋆⋆], and Michael Wiener[3]

[1] Katholieke Universiteit Leuven, ESAT/COSIC, K. Mercierlaan 94, 3001 Heverlee, Belgium ({erik.dewin,bart.preneel}@esat.kuleuven.ac.be)
[2] Queen's University, Department of Electrical and Computer Engineering, Kingston, Ontario, K7L 3N6, Canada (cf744@freenet.carleton.ca)
[3] Entrust Technologies, 750 Heron Road, Ottawa (Ontario) K1V 1A7, Canada (wiener@entrust.com)

Abstract. This paper describes a fast software implementation of the elliptic curve version of DSA, as specified in draft standard documents ANSI X9.62 and IEEE P1363. We did the implementations for the fields GF(2^n), using a standard basis, and GF(p). We discuss various design decisions that have to be made for the operations in the underlying field and the operations on elliptic curve points. In particular, we conclude that it is a good idea to use projective coordinates for GF(p), but not for GF(2^n). We also extend a number of exponentiation algorithms, that result in considerable speed gains for DSA, to ECDSA, using a signed binary representation. Finally, we present timing results for both types of fields on a PPro-200 based PC, for a C/C++ implementation with small assembly-language optimizations, and make comparisons to other signature algorithms, such as RSA and DSA. We conclude that for practical sizes of fields and moduli, GF(p) is roughly twice as fast as GF(2^n). Furthermore, the speed of ECDSA over GF(p) is similar to the speed of DSA; it is approximately 7 times faster than RSA for signing, and 40 times slower than RSA for verification (with public exponent 3).

1 Introduction

Elliptic curve public key cryptosystems (ECPKCs) were proposed independently by Victor Miller [M85a] and Neil Koblitz [K87] in the mid-eighties, but it is only recently that they are starting to be used in commercial systems. See [M93] for an introduction to practical aspects of public key cryptosystems based on elliptic curves. The elliptic curve discrete logarithm problem (ECDLP) has been studied for several years now, and no significant weaknesses have been found, although some special instances of it have been broken [MOV93], [S97a].

⋆ F.W.O.-Flanders research assistant, sponsored by the Fund for Scientific Research – Flanders. Most of the work presented in this paper was done during an internship with Entrust Technologies in Ottawa, Canada.
⋆⋆ F.W.O.-Flanders postdoctoral researcher, sponsored by the Fund for Scientific Research – Flanders.

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

A number of publications discuss software implementations of ECPKCs. [HMV92] is probably the earliest, and uses the field GF(2^n), where the field elements are represented in an optimal normal basis [MOVW88] or as polynomials over the subfield GF(2^8). [SOOS95] uses a standard basis for GF(2^n), where the irreducible field polynomial is a trinomial. [DBV+96] and [GP97] represent the elements of GF(2^n) as polynomials over the subfield GF(2^{16}). Few comparisons of ECPKCs to other public key cryptosystems are available; only [SOOS95] compares Diffie-Hellman key agreement using elliptic curves over GF(2^n) to its counterpart using large integer numbers, and concludes that the elliptic curve-based version is several times faster, the exact ratio depending on the platform and the amount of optimization used. As far as we know, [MOC97] is the only implementation of ECPKCs over GF(p) that has been reported, and no comparisons have been made between elliptic curves over GF(2^n) and over GF(p).

In this paper, we present an implementation of a signature scheme based on elliptic curves. The signature scheme used is elliptic curve DSA (ECDSA), as defined in the ANSI X9.62 draft standard and the IEEE P1363 draft standard. We consider curves both over GF(2^n) and GF(p), in each case using curves that are specified in ANSI X9.62.

The remaining part of this paper is organized as follows. Section 2 gives more background on elliptic curves, elliptic curve public key cryptosystems, and related standards. Sections 3 and 4 discuss implementation considerations that are specific to GF(p) and GF(2^n) respectively. Section 5 discusses issues related to operations on elliptic curve points, operations that are common to both GF(p) and GF(2^n). The overall timing results and comparisons to other public key cryptosystems appear in Section 6. A number of topics for further work and research are given in Section 7.

2 Elliptic curve cryptosystems

Elliptic curves have been studied by mathematicians since long before they were used in cryptography. Apart from their use for public key cryptosystems, they formed the basis of the elliptic curve factoring method [L87] and of several methods for primality proving, e.g. [AM93]. Recently, they were an important tool in the proof of Fermat's last theorem.

An elliptic curve is the set of solutions of a *Weierstrass equation* over a mathematical structure, usually a field. For cryptographic purposes, this field is mostly a *finite* field of the form either GF(p) or GF(2^n). In these particular cases, the Weierstrass equation can be reduced to the following simpler forms:

$$y^2 = x^3 + ax + b \text{ over GF}(p), \text{ with } a, b \in \text{GF}(p) \text{ and } 4a^3 + 27b^2 \neq 0 \ ;$$
$$y^2 + xy = x^3 + ax^2 + b \text{ over GF}(2^n), \text{ with } a, b \in \text{GF}(2^n) \text{ and } b \neq 0 \ .$$

If the formal *point at infinity* $\mathcal{O}$ is added to the set of solutions, an addition operation can be defined, and this turns the set into a group. The addition operation is defined as follows. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points on the elliptic curve, neither the point at infinity.

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

Over GF(p): The inverse of a point P_1 is $-P_1 = (x_1, -y_1)$. If $P_2 \neq -P_1$, then $P_1 + P_2 = P_3 = (x_3, y_3)$, with

$$x_3 = \lambda^2 - x_1 - x_2 \ ,$$
$$y_3 = \lambda(x_1 - x_3) - y_1 \ ,$$

and

$$\lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} \text{ if } P_1 \neq P_2 \ , \\ \dfrac{3x_1^2 + a}{2y_1} \text{ if } P_1 = P_2 \ . \end{cases}$$

Over GF(2^n): The inverse of a point P_1 is $-P_1 = (x_1, y_1 + x_1)$. If $P_2 \neq -P_1$, then $P_1 + P_2 = P_3 = (x_3, y_3)$ with

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \ ,$$
$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \ ,$$

and

$$\lambda = \begin{cases} \dfrac{y_2 + y_1}{x_2 + x_1} \text{ if } P_1 \neq P_2 \ , \\ x_1 + \dfrac{y_1}{x_1} \text{ if } P_1 = P_2 \ . \end{cases}$$

For both fields we have the following formulas for the cases where the point at infinity is involved: $P_1 + (-P_1) = \mathcal{O}, \mathcal{O} + P_1 = P_1 + \mathcal{O} = P_1$ and $\mathcal{O} + \mathcal{O} = \mathcal{O}$.

The basic assumption of elliptic curve public key cryptosystems is that the discrete logarithm problem in the elliptic curve group (ECDLP) is a hard problem. Hence all public key cryptographic primitives based on the discrete logarithm over the integers modulo a prime can be translated to an equivalent primitive based on the ECDLP. Moreover, the ECDLP is currently considered to be harder than the integer DLP. Therefore, the sizes of fields, keys, and other parameters can be chosen considerably smaller for elliptic curve based systems; typical field sizes are between 160 and 200 bits. This can be especially advantageous in systems where resources such as memory and/or computing power are limited, but even where this is not the case, ECPKCs turn out to be competitive to other public key cryptosystems such as RSA and DSA.

An important condition for the practical usefulness of ECPKCs is that we can efficiently implement the point multiplication operation, which is the repeated group operation, and the equivalent of exponentiation in systems based on the discrete logarithm problem for integers modulo a prime. As became clear from the definition above, the elliptic curve group operation can be expressed in terms of a number of operations in the underlying field. For all cases where the point at infinity is not involved, we see that for the calculation of one group operation, we need 1 field inversion, 2 general field multiplications, 1 or 2 field squarings, a number of field additions or subtractions, and a number of multiplications by a fixed small constant. For the case GF(2^n), we will see that only field inversions

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

and multiplications need to be counted, since the other operations are much faster and their share in the overall time for a group operation is negligible. For GF(p), the time needed for a squaring is of the same order of magnitude as the time needed for a multiplication, so we have to take into account the squarings as well; the number of squarings is 1 for a general point addition and 2 for a point doubling (i.e. the case where $P_1 = P_2$).

An important design decision is how the field elements are represented. We discuss this issue for each field separately in Sections 3 and 4.

A number of standardization bodies have started initiatives to standardize ECPKCs, among them are ANSI, IEEE, ISO, IETF. Most standards are still drafts, but are expected to be approved in the near future. The specified schemes include signature schemes, encryption schemes, and key agreement schemes. ECDSA is specified in ANSI X9.62 and IEEE P1363; both descriptions are almost identical. We based our implementation on the most recent draft documents we had available, i.e. [A97] and [I97].

Both [A97] and [I97] provide the option to apply point compression to points on an elliptic curve, in order to reduce storage requirements or bandwidth. The basic idea is that specifying the two coordinates of a point is unnecessary, since the fact that they satisfy the curve equation provides redundancy. More specifically, if the x-coordinate is known, at most two values of y are possible, and they can be computed by solving a quadratic equation. One extra bit of information allows to distinguish between the two values of y; this means that an elliptic curve point needs only slightly more storage space than an element of the underlying field. We do not discuss point compression in the rest of the paper since the cost is small compared to the cost of the overall signing and verification operations.

3 Elliptic curves over GF(p)

In this section we describe implementation issues that are specific to curves over the field GF(p). The issues that apply to both GF(p) and GF(2^n) will be discussed in Section 5.

3.1 Representation of field elements

For GF(p), the most obvious way to represent the elements is as numbers in the range $[0, p-1]$, where each residue class is represented by its member in that range. Yet, it is not the only way. Since we will be using modular multiplications and squarings, we might consider representing the elements as Montgomery residues [M85b]. This only influences the inversion operation, since the inverse of a Montgomery residue is not the Montgomery residue of the inverse, i.e. the inverse operation does not commute with taking the Montgomery residue. Hence an extra transformation is needed, but this problem can be alleviated by using the algorithm described in [K95a] that computes the Montgomery inverse. Moreover, if projective coordinates are used (see Section 3.4), very few inverse

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

operations are needed anyway. Despite all this, we decided not to use the Montgomery representation for the simple reason that in our implementation the difference in speed between Montgomery and Barrett [B87] reduction is negligibly small. This also saves us the hassle of having to implement a special inverse algorithm and converting between two representations. However, in some cases the representation as Montgomery residues could be advantageous.

3.2 Field multiplication and squaring

For field multiplication and squaring, we started from our own implementation in C of well-known algorithms for operations on multi-precision numbers, see e.g. [K81]. Since standard C does not support the full capabilities of modern PCs for integer multiplication and division (i.e. 32-bit×32-bit→64-bit and 64-bit/32-bit→32-bit), we used a number of small assembly language macros to make these available. As discussed in Section 3.1, we use a Barrett-like modular reduction algorithm.

3.3 Field inversion

In most public key cryptosystems that are not based on elliptic curves, the time spent computing modular inverses is negligible compared to the time needed for modular exponentiation. Therefore, in many implementations not much effort has been spent on optimizing the modular inverse algorithm. However, in a straightforward implementation of the equations in Section 2, every single group operation needs to compute one modular inverse, and it turns out that this is where most of the execution time goes. It therefore is worthwhile to give some more thought to the optimization of this operation.

We compared a number of algorithms, mostly variants of the extended version of Euclid's algorithm. The best results were obtained with an algorithm that is based on the Montgomery inverse algorithm [K95a], after speeding it up by applying some extra heuristics and using the same assembly language macros as for multiplication and squaring. For lack of space, we cannot discuss the algorithm in detail here. It suffices to state that we were able to considerably improve the inversion operation, but we still found a ratio of 23 between the time for an inversion and a multiplication in a field with a 192-bit modulus.

3.4 Projective coordinates

With a ratio of 23 between inversion and multiplication, it is clear that the former operation will be the major bottleneck of the implementation. Fortunately, there are ways to circumvent this problem, and they lie in the possibility of using different ways to specify the group operation. An alternative definition, that is explicitly specified in the appendices to [I97], uses projective coordinates. In this representation, the elliptic curve equation has 3 variables, and a point has 3 coordinates (x, y, z), but any point with coordinates $(\lambda^2 x, \lambda^3 y, \lambda z)$ for arbitrary

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

$\lambda \neq 0$ is considered equal to the former. In fact, this can be thought of as keeping the denominator of the equations for the group operation in a separate variable, and postponing the actual inversion operation until the x- or y-coordinates are really needed, for instance at the end of a point multiplication.

The drawback of projective coordinates is that a group operation involves considerably more field multiplications. In [I97], projective formulas are given that allow a point doubling to be computed using 10 field multiplications in the general case, and 8 if the curve parameter a is 3 less than the modulus. A point addition requires 16 multiplications in the general case, and only 11 if one of the points has a z-coordinate equal to 1. On the whole, assuming that an inversion takes the time of approximately 23 multiplications, we can save roughly between 10 and 19 multiplication times per group operation.

4 Elliptic curves over GF(2^n)

In this section we describe implementation issues that are specific to curves over the field GF(2^n). The issues that apply to both GF(p) and GF(2^n) will be discussed in Section 5.

4.1 Representation of field elements

For GF(2^n), a number of representations of the field elements are known and each of them has its specific advantages. The most well known representation is the *standard basis* representation, used for instance in [SOOS95]. Field elements are represented as binary polynomials modulo an irreducible binary polynomial of degree n. Standard basis implementations can be made more efficient if an irreducible polynomial with low Hamming weight and no terms of high degree is chosen, such as a trinomial or a pentanomial. At least one of these can be found for any value of n.

Another well known representation uses an *optimal normal basis* [MOVW88]. This basis gives rise to elegant hardware implementations, but for software, our experience is that a standard basis is more efficient.

A third representation (see e.g. [HMV92], [DBV+96] or [GP97]) represents elements of the field as polynomials over a subfield of the form GF(2^r), where r is a divisor of n. This representation enables efficient implementations, but limits the possible values of n to multiples of r. This is not so much an implementation issue, since we can make r small enough that the number of possible values of n is still sufficiently large. But the fact that these fields have some extra structure, consisting of a fairly large subfield, could reduce the security in the sense that the ECDLP over these fields might turn out to be easier to break.

Although there currently is no indication that the latter fields are less secure, we wanted to avoid the risk by choosing a prime n. And since optimal normal bases seem to be slower in software, we opted for a standard basis representation using trinomials or pentanomials. This representation is well specified in both [A97] and [I97].

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

4.2 Field multiplication and squaring

The algorithms for multiplication and squaring in a standard basis, as well as algorithms for reduction modulo a trinomial or pentanomial, are described in [SOOS95]. Contrary to GF(p), no assembly language was used, because most microprocessors do not have a special instruction for multiplying binary polynomials. While this may seem to result in a biased comparison between both kinds of fields, the situation in a practical application is likely to be similar, hence our comparisons are practically relevant.

Note that the squaring operation is much more efficient than multiplication, because GF(2^n) has characteristic two, so that all the cross-terms vanish.

4.3 Field inversion

The almost inverse algorithm [SOOS95] is the fastest known algorithm for computing modular inverses of binary polynomials. With a suitable choice of the field polynomial, the inversion time is approximately 3 times longer than the multiplication time.

At the end of the algorithm, a Montgomery-like reduction is necessary to convert the almost inverse to the real inverse. This reduction is fast if the irreducible field polynomial has low Hamming weight and has no terms of low degree (smaller than the word size of the processor), except for the constant term. Unfortunately, most of the field polynomials specified in ANSI X9.62 do have terms of low degree. This increases the timings of the almost inverse algorithm by up to 30%. Therefore, we conclude that the choice of polynomials in ANSI X9.62 is rather unfortunate, and may be revised if that is practically feasible.

This problem can be circumvented by converting the field elements and elliptic curve points from a representation based on a standardized polynomial to an internal representation based on a polynomial with better properties. We did not implement the conversion yet, but we give timing results using both a polynomial from the standard and a more suitable polynomial.

Because the ratio between field inversion and field multiplication is rather low, the use of projective coordinates brings no benefit for GF(2^n) in a standard basis representation.

4.4 Basis Conversion

Although a single basis may be chosen for a program's internal representation of field elements, it is important for interoperability with other implementations that an efficient method of converting between the chosen representation and the others exist. This is the case for the bases already discussed; the procedure involves finding a root (in the target basis) of the field polynomial of the original basis. The field element in the target basis is then calculated as the linear combination of powers of that root. Details are provided in [A97] for conversion between standard and optimal normal bases. In practice, the calculation of

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

the root is expensive, so the roots are tabulated and the required powers are calculated during the first conversion.

Apart from interoperability, basis conversion is also useful from an efficiency point of view, for example for field inversion (see Section 4.3).

5 Operations on elliptic curve points

The basic group operations can be implemented in a straightforward way in terms of the field operations discussed in Sections 3 and 4. However, the core operation of ECPKCs is the *repeated* group operation, i.e. the multiplication of a point by an integer, and this operation deserves some more thought. It is the equivalent of modular exponentiation for integer DLP-based systems, and is therefore also referred to as elliptic curve exponentiation, and the multiplier is sometimes called the exponent. We will use both terms interchangeably; we are confident that this will not cause confusion since strictly speaking there exists no such thing as elliptic curve exponentiation.

Many authors have discussed fast ways to do exponentiation under various conditions; [G96] gives a concise overview. Most of these algorithms can be extended to the point multiplication in an elliptic curve group. However, the elliptic curve group has the interesting property that the inverse of a point is extremely efficient to compute (see Section 2). This allows for some extra optimizations [R60]. On the other hand, exponents in an elliptic curve system are generally much shorter than in other systems such as RSA. Some optimizations described in the literature may only be advantageous for exponents above certain lengths, and may not be worthwhile for elliptic curves.

In the next paragraphs, we discuss point multiplication for a number of cases that are relevant to ECDSA. The algorithms are mostly based on known algorithms for exponentiation, but we adapt them in order to make better use of the parameters of the elliptic curve case. Before that, we will discuss some issues related to the representation of the exponent.

5.1 Representation of exponents

The binary representation can be considered as the generic representation for exponents, because it is the basis for the square-and-multiply algorithm. This algorithm is discussed in [K81, p. 442] for instance, and gives an extremely simple and relatively efficient way to find addition chains. It has been improved upon in a number of ways depending on the context, e.g. by using windowed methods, or precomputation, but the binary representation remains the basis of many practical implementations.

For elliptic curve exponentiation, the binary representation can still be used, but a *signed binary* representation, where each bit has a sign, seems more appropriate. A negative bit is processed similarly as a positive bit, but uses the inverse of the point, which can easily be calculated and used in the course of an

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

exponentiation. It is important to note that this representation is not unique, e.g. $1000\bar{1}$ and 1111 both represent the number 15 ($\bar{1}$ stands for a negative bit).

In [MO90], an algorithm is proposed to convert from a non-signed to a particular signed representation. The result has the so-called *non-adjacent form* (NAF); this means that of any two adjacent bits, at least one must be zero. An interesting property of the NAF representation is that it is unique. Also, for a random exponent, the expected fraction of non-zero bits is 1/3, as opposed to 1/2 for a binary representation. This results in an 11% speed-up on average for the standard square-and-multiply algorithm. As we will see, the use of the NAF can speed up windowed techniques as well.

Although the recoding algorithm in [MO90] looks a little involved, the signed binary NAF of a number e can be computed easily as follows: subtract e from $3e$, replacing the borrow mechanism by the rule $0 - 1 = \bar{1}$, and then discard the least significant bit.

Alternative signed binary representations have been proposed in [KT92] and [MOC97]. These representations have better properties with respect to windowed exponentiation techniques. However, we will see that in comparing different representations, it is important to take into account the number of precomputations. It is an open problem what the best signed binary representation for windowed techniques is.

To analyze the expected number of operations for a point multiplication, we need an estimate of the expected length of a run of zeros, since this has an impact on the expected number of additions. According to [KT92], this average length is 4/3 for the signed binary NAF and 3/2 for the improved method they propose. In [MOC97], an algorithm is proposed that results in an average zero-run length of 2. The binary representation has an average zero-run length of 1 [K95b].

5.2 General point multiplication

The square-and-multiply (or double-and-add in additive notation) algorithm can easily be extended to a double-and-add/subtract algorithm based on signed binary NAF. The expected improvement is roughly 11% [MO90].

Other algorithms, such as the sliding window technique, can be extended to the signed binary NAF representation as well. We will first give an example for a particular window size, and then generalize the results to arbitrary window sizes.

With a window size w of 4 bits, the only windows that can occur are 1000, 1001, 1010, $100\overline{1}$, $10\overline{1}0$, plus their counterparts with the signs of all bits reversed. The values associated to the latter can easily be computed as the negative of the precomputed values associated to the former. All other window values are excluded because of the NAF property. Denoting the point to be multiplied by P, this means that we only have to precompute and store $6P$, $7P$, $8P$, $9P$ and $10P$; the other values can be obtained from these by taking the negative. The precomputation can be done in 7 operations using the addition sequence $1, 2, 4, 6, 7, 8, 9, 10$. This can even be reduced to 5 operations if trailing zeros are

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

handled such as in alg. 14.85 of [MvV97]. In this case, only $3P$, $5P$, $7P$ and $9P$ need to be precomputed.

If we consider the window size w as a parameter, the average number of operations for a complete point multiplication is

$$C(w) + \lambda + 2 - w + \frac{\lambda + 1 - w/2}{w + 4/3} \ , \tag{1}$$

where $\lambda = \lfloor \log_2(k) \rfloor$ (denoting the exponent by k), 4/3 is the average zero-run-length, and $C(w)$ is the number of operations needed for the precomputation. The expression for $C(w)$ for a signed binary NAF is slightly more complex than for the binary case:

$$C(w) = \frac{2^w - (-1)^w}{3} \ .$$

The algorithm described here was considered in [KT92], and in the same paper, an improvement was proposed, consisting of an alternative, slightly more complex, algorithm to convert from binary to signed binary representation. This results in an increased average length of zero runs and a reduced number of operations in the course of the algorithm. As an example, consider the bit string 00111100 as part of an exponent; this is replaced by $0100\overline{1}00$ in the NAF. With a window size of 4, two add/subtract operations are needed to handle the NAF of this bit string, whereas the original form potentially needs one addition, depending on the other exponent bits; hence it is better not to do the substitution. The expected number of operations of the improved algorithm is [KT92]:

$$(\lambda + 2.75 - w) + \frac{\lambda + 1.25}{w + 1.5} + 2^{w-1} - 1 \ . \tag{2}$$

When comparing the number of operations given by (1) and (2) for exponents up to 2000 bits, we find that the latter algorithm needs in fact more operations than the former, contrary to the conclusion in [KT92]. This is probably due to an overestimation of the cost of precomputation $C(w)$ in (1): because of the NAF property, a considerable number of values do not have to be precomputed since they cannot occur. Since the algorithm proposed in [KT92] does not produce a NAF, we see no way to obtain comparable savings for the precomputation step.

We used the first algorithm in our implementation. The optimal window size is 4 for exponents up to roughly 170 bits, 5 for the range 170–420 bits, 6 for the range 420–1290 bits and 7 for 1290 bits up to well above 2000 bits. Comparing to a sliding window technique based on the binary representation, we gain approximately 2.6% for 200-bit exponents, decreasing to only 1.3% for 2000 bits.

In a recent paper [MOC97], an even better recoding algorithm is proposed, resulting in an average zero-run length of 2. To our understanding, there is no restriction on the values of the windows, and the number of values to be precomputed is $2^{w-1} - 1$, as in (2). When we calculate the expected number of operations under these assumptions, we find that the difference with the signed binary NAF remains under a fraction of a percent for exponent lengths up to

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

over 2000 bits. In that range, there are alternating subranges for which signed binary NAF is better than [MOC97] and vice-versa.

Note that in the estimates of the number of operations, no distinction was made between additions and doublings. For GF(2^n), this is a good approximation, since both operations are almost equally fast. However, for GF(p) with projective coordinates, a typical doubling is 25% faster than a typical addition, so an accurate estimate of the number of operations should make a distinction between them. Fortunately, this distinction has very little influence on the optimal window size, since the number of doublings depends only lightly on it.

5.3 ECDSA key generation and signing operation

Most of the time needed for key generation and signing is typically taken by the multiplication of the EC group generator by a random number. The EC group and generator are typically known ahead of time; therefore, we can afford to do some precomputation at initialization time in order to obtain a faster signing operation. A number of algorithms for exponentiation with a fixed generator have been described in [BGMW92]. We use a rather simple one, which is also described in algorithm 14.109 of [MvV97]. We denote the group generator by P and use the additive notation. After choosing a basis b, we precompute the products $b^i P$ for all values of i up to a certain bound t so that all multipliers will be smaller than b^t. Then the algorithm does a point multiplication in at most $t + h - 2$ group operations, where h is the maximum value of the digits in the b-ary representation of the exponent.

To avoid doing basis conversions, we choose $b = 2^w$, which essentially results in a windowed method with window size w. If we use a binary representation for the exponent, $h = b - 1$. However, if we use the signed binary NAF, a number of high values of the window are impossible and h is reduced to

$$h = 2\frac{2^w - 1}{3} \qquad \text{for even } w \ ,$$

$$h = \frac{2^{w+1} - 1}{3} \qquad \text{for odd } w \ .$$

For the curves and field sizes used for the timings, using the NAF reduces the signature time by almost 10%.

Note that the algorithm we implemented is not the best algorithm known. The signing time can be reduced even more using a slightly more advanced recoding algorithm from [BGMW92], which has $h = 2^{w-1}$. With the parameters used for our timings, this would result in an additional 5% gain. Recently, an algorithm was proposed [MOC97] that gives better results for elliptic curves over GF(p). The algorithm is substantially different from the algorithms discussed in [BGMW92]; it trades point additions for doublings, which are more efficient when projective coordinates are used (see Section 3.4).

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

5.4 ECDSA verification operation

Both the DSA and the ECDSA operation require the computation of a *simultaneous multiple point multiplication*, i.e., a group element of the form $k_1P_1+k_2P_2$, where P_1 and P_2 are elements of the group and k_1 and k_2 are integers. Algorithm 14.88 of [MvV97] gives a way to compute this in an interleaved way, rather than by calculating the two point multiplications separately and adding the result. If this algorithm is combined with a sliding window technique, we obtain an algorithm that is only 20%-25% slower than a single point multiplication. The optimal window length is 2 for exponents up to at least 500 bits. From simulations, we estimate that the average length of a zero run is approximately 0.6. The number of operations is given by a formula similar to (1).

6 Timings and comparison

We timed our implementation for a number of example curves from the current draft of ANSI X9.62. For GF(p) we used a modulus of 192 bits. The curve parameter a of the example curve is 3 less than the modulus, allowing for a faster projective doubling operation. For GF(2^n), we did timings for 2 trinomials of degree 191, one that is specified in the standard, and one that has better properties with respect to the reduction step of the almost inverse algorithm. For the latter, we did not use a curve from the standard.

The timings were done on a PPro200-based PC with Windows NT 4.0 using MSVC 4.2 and maximal optimization. The code for RSA and DSA was written in C, using some small macros in assembly language. The elliptic curve code was mainly written in C++; for GF(p) the same multi-precision routines in C were called as for RSA and DSA.

Table 1 gives timing results for the field operations and the elliptic curve group operations for both GF(p) and GF(2^n). The computation of inverses is clearly more expensive over GF(p), but this is largely compensated for by the faster multiplication, since projective coordinates can be used.

Table 1. Timings for field operations over GF(p) and GF(2^n). The field size is approximately 191 bits for both. For GF(2^n), two timings are given, one using a trinomial specified in ANSI X9.62 and the other using a trinomial with better properties with respect to the almost inverse algorithm. All times in μs.

	GF(p)	GF(2^n), standard trinomial	GF(2^n), improved trinomial
addition	1.6	0.6	0.6
multiplication	7.8	39	39
squaring	7.6	2.6	2.6
inverse	180	159	126
EC addition	103	242	215
EC double	76	246	220

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

Table 2 gives timing results for the overall key generation, signing, and verification operations for ECDSA, RSA and DSA, as well as for general point multiplication on an elliptic curve. For DSA and ECDSA, we assumed that the underlying group is the same for all users; if this is not the case, the key generation time has to be augmented by the time needed to generate an appropriate group (such as prime generation, point counting on an elliptic curve, etc.).

Table 2. Comparison of ECDSA to other signature algorithms. For EC, the field size is approximately 191 bits. The modulus for RSA and DSA is 1024 bits long; the RSA public exponent is 3. All times in ms, unless otherwise indicated.

	ECDSA GF(2^n) standard trin.	ECDSA GF(2^n) improved trin.	ECDSA GF(p)	RSA	DSA
key generation	13.0	11.7	5.5	1s	22.7
signature	13.3	11.3	6.3	43.3	23.6
verification	68	60	26	0.65	28.3
general point multipl.	56	50	21.1		

The modulus for both RSA and DSA is 1024 bits long. There is no general consensus about the relative security levels of EC, RSA, and DSA as a function of the size of the parameters. It is probably safe to state that EC with a group size of 190 bits is slightly stronger than RSA or DSA with a 1024-bit modulus. The RSA public exponent is 3. Note that the DSA implementation does not use precomputation for the key generation and signing operation, whereas ECDSA does.

7 Further work

There are still a number of potential optimizations we have not used in our implementation.

For GF(2^n), *anomalous curves* could be used [K91]. In [S97b], an algorithm is proposed that requires less than $\lambda/3$ elliptic curve additions and a number of field squarings, the latter being almost for free in GF(2^n). This would be particularly interesting to speed up the verification operation. Note that anomalous curves over GF(p) should be avoided for cryptographic use [S97a]; for anomalous curves over GF(2^n) no particular weaknesses have been found.

For key and signature generation, the optimizations described at the end of Section 5.3 could be implemented. Using an advanced technique from [BGMW92] might further improve the speed of these operations.

In [GP97], an improved point multiplication algorithm is described, based on a more efficient way to repeatedly double a point by trading inversions for multiplications. The paper only discusses the GF(2^n) case, and is currently not advantageous for our implementation because the inversion is relatively fast. However, a similar idea can probably be applied to GF(p), and there the benefit

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

could be more important because of the fast field multiplication. Note that this idea cannot be combined with projective coordinates; more work is needed to determine which of the two results in the fastest implementation.

References

[A97] ANSI X9.62-199x: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), June 11, 1997.

[AM93] A. Atkin and F. Morain, "Elliptic curves and primality proving," *Mathematics of Computation*, Vol. 61 (1993), pp. 29–68.

[B87] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," *Advances in Cryptology, Proc. Crypto'86, LNCS 263*, A. Odlyzko, Ed., Springer-Verlag, 1987, pp. 311–323.

[BGMW92] E. Brickell, D. Gordon, K. McCurley and D. Wilson, "Fast exponentiation with precomputation," *Advances in Cryptology, Proc. Eurocrypt'92, LNCS 658*, R.A. Rueppel, Ed., Springer-Verlag, 1993, pp. 200–207.

[DBV+96] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem and J. Vandewalle, "A fast software implementation for arithmetic operations in GF(2^n)," *Advances in Cryptology, Proc. Asiacrypt'96, LNCS 1163*, K. Kim and T. Matsumoto, Eds., Springer-Verlag, 1996, pp. 65–76.

[G96] D. Gordon, "A survey of fast exponentiation methods," draft, 1996.

[GP97] J. Guajardo and C. Paar, "Efficient algorithms for elliptic curve cryptosystems," *Advances in Cryptology, Proc. Crypto'97, LNCS 1294*, B. Kaliski, Ed., Springer-Verlag, 1997, pp. 342–356.

[HMV92] G. Harper, A. Menezes and S. Vanstone, "Public key cryptosystems with very small key length," *Advances in Cryptology, Proc. Eurocrypt'92, LNCS 658*, R.A. Rueppel, Ed., Springer-Verlag, 1993, pp. 163–173.

[I97] IEEE P1363: Editorial Contribution to Standard for Public Key Cryptography, August 18, 1997.

[K95a] B. Kaliski Jr., "The Montgomery inverse and its applications," *IEEE Transactions on Computers*, Vol. 44, no. 8 (1995), pp. 1064–1065.

[K81] D. Knuth, *The art of computer programming, Vol. 2, Semi-numerical Algorithms,* 2^{nd} *Edition*, Addison-Wesley, Reading, Mass., 1981.

[K87] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, Vol. 48, no. 177 (1987), pp. 203–209.

[K91] N. Koblitz, "CM-curves with good cryptographic properties," *Advances in Cryptology, Proc. Crypto'91, LNCS 576*, J. Feigenbaum, Ed., Springer-Verlag, 1997, pp. 279–287.

[K95b] C. Koç, "Analysis of sliding window techniques for exponentiation," *Computers Math. Applic.*, Vol. 30, no. 10 (1995), pp. 17–24.

[KT92] K. Koyama and Y. Tsuruoka, "Speeding up elliptic cryptosystems by using a signed binary window method," *Advances in Cryptology, Proc. Crypto'92, LNCS 740*, E. Brickell, Ed., Springer-Verlag, 1993, pp. 345–357.

[L87] H.W. Lenstra Jr., "Factoring integers with elliptic curves," *Annals of Mathematics*, Vol. 126 (1987), pp. 649–673.

[M93] A. Menezes, *Elliptic curve public key cryptosystems*, Kluwer Academic Publishers, 1993.

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.

[MOV93] A. Menezes, T. Okamoto and S. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field," *IEEE Transactions on Information Theory*, Vol. 39 (1993), pp. 1639–1646.

[MvV97] A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997.

[M85a] V.S. Miller, "Use of elliptic curves in cryptography," *Advances in Cryptology Proc. Crypto'85, LNCS 218*, H.C. Williams, Ed., Springer-Verlag, 1985, pp. 417–426.

[MOC97] A. Miyaji, T. Ono and H. Cohen, "Efficient elliptic curve exponentiation," *Proceedings of ICICS'97, LNCS 1334*, Y. Han, T. Okamoto and S. Qing, Eds., Springer-Verlag, 1997, pp. 282–290.

[M85b] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, Vol. 44 (1985), pp. 519–521.

[MO90] F. Morain and J. Olivos, "Speeding up the computations on an elliptic curve using addition-subtraction chains," *Informatique Théorique et Applications*, Vol. 24, pp. 531–543, 1990.

[MOVW88] R. Mullin, I. Onyszchuk, S. Vanstone and R. Wilson, "Optimal normal bases in GF(p^n)," *Discrete Applied Mathematics*, Vol. 22 (1988/1989), pp. 149–161.

[R60] G. Reitwiesner, "Binary arithmetic," *Advances in Computers*, Vol. 1 (1960), pp. 231–308

[SOOS95] R. Schroeppel, H. Orman, S. O'Malley and O. Spatscheck, "Fast key exchange with elliptic curve systems," *Advances in Cryptology, Proc. Crypto'95, LNCS 963*, D. Coppersmith, Ed., Springer-Verlag, 1995, pp. 43–56.

[S97a] N. Smart, "Elliptic Curve Discrete Logarithms," message to newsgroup sci.math.research. no. 3430BAB8.4878@hplb.hpl.hp.com, Sept. 30 1997.

[S97b] J. Solinas, "An improved algorithm for arithmetic on a family of elliptic curves," *Advances in Cryptology, Proc. Crypto'97, LNCS 1294*, B. Kaliski, Ed., Springer-Verlag, 1997, pp. 357–371.

Appeared in *Algorithmic Number Theory, 3rd International Symposium, ANTS 1998*, Lecture Notes in Computer Science 1423, J. Buhler (ed.), Springer-Verlag, pp. 252–266, 1998.