Implementing an efficient elliptic curve cryptosystem over $GF(p)$ on a smart card

Yvonne Hitchcock[*] Edward Dawson[†]
Andrew Clark[‡] Paul Montague[§]

(Received 1 June 2001; revised 24 October 2002)

Abstract

Elliptic curve cryptosystems (ECCs) are becoming more popular because of the reduced number of key bits required in comparison to other cryptosystems (for example, a 160 bit ECC has roughly the same security as 1024 bit RSA). ECCs are especially suited to smart cards because of the limited memory and computational power available on these devices.

[*]Information Security Research Centre, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia. mailto:y.hitchcock@qut.edu.au

[†]as above. mailto:e.dawson@qut.edu.au

[‡]as above. mailto:a.clark@qut.edu.au

[§]Motorola Australia Software Centre, 2 Second Ave, Mawson Lakes, SA 5095, Australia. mailto:pmontagu@asc.corp.mot.com

[0]See http://anziamj.austms.org.au/V44/CTAC2001/Hitc for this article, © Austral. Mathematical Soc. 2003. Published 1 April 2003. ISSN 1446-8735

This paper discusses an optimized implementation of the elliptic curve Digital Signature Algorithm implemented over the field $GF(p)$ on a Motorola smart card. Algorithms for point addition, point doubling and scalar multiplication are compared according to their timings. The effects of different memory usage, code size and speed tradeoffs which were considered during the implementation are discussed. Also, optimized point addition and doubling algorithms are presented.

Contents

1 Introduction

Elliptic curves were first proposed as a basis for public key cryptography in the mid 1980s independently by Koblitz [11] and Miller [14]. Elliptic curves provide a public key cryptosystem based on the difficulty of the elliptic curve discrete logarithm problem (defined later in this section), which is so called because of its similarity to the discrete logarithm problem (DLP) over the integers modulo a prime p. This similarity means that most cryptographic procedures carried out using a cryptosystem based on the DLP over the integers modulo p can also be carried out in an elliptic curve cryptosystem. Another benefit of ECCs is that they can use a much shorter key length than other public key cryptosystems to provide an equivalent level of security. For example, 160 bit elliptic curve cryptosystems (ECCs) are believed to provide about the same level of security as 1024 bit RSA [7, p.51]. Also, the rate at which ECC key sizes increase in order to obtain increased security is much slower than the rate at which integer based discrete logarithm (DL) or RSA key sizes must be increased for the same increase in security. ECCs can also provide a faster implementation than RSA or DL systems, and use less bandwidth and power [9]. These issues are crucial in lightweight applications such as smart cards.

In the last few years, confidence in the security of ECCs has also risen, to the point where they have now been included or proposed for inclusion in internationally recognized standards (specifically IEEE Std 1363-2000, WAP (Wireless Application Protocol), ANSI X9.62, ANSI X9.63 and ISO CD 14888-3). Thus elliptic curve cryptography is set to become an integral part of lightweight applications in the immediate future.

An elliptic curve over a Galois field with p elements, $GF(p)$ [12], where p is prime and $p > 3$ may be defined as the points (x, y)

satisfying the curve equation $E : y^2 = x^3 + ax + b \pmod{p}$, where a and b are constants satisfying $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$. In addition to the points satisfying the curve equation E, a point at infinity, ϕ, is also defined. With a suitable definition of addition and doubling of points [3], this enables the points of an elliptic curve to form a group with addition and doubling of points being the group operation, and the point at infinity being the identity element. We then further define scalar multiplication of a point P by a scalar k as being the result of adding the point P to itself k times $(kP = P + P + \cdots + P \quad (k \text{ times}))$. The elliptic curve discrete logarithm problem is then defined as follows: Given the prime modulus p, the curve constants a and b and two points P and Q, find a scalar k such that $Q = kP$. This problem is infeasible for secure elliptic curves, and thus scalar multiplication is the basic cryptographic operation of an elliptic curve.

In order to achieve an efficient implementation, firstly efficient field arithmetic (modular addition, subtraction, multiplication and inversion) must be available. These operations are then used in the algorithms for addition and doubling of points. In turn, the addition and doubling operations must be efficient, in order for the scalar multiplication which uses them to be efficient. It is possible to add and double points in various coordinate systems. The choice of coordinate system has a considerable impact on the final speed of the scalar multiplication operation.

In this paper we investigate the efficient implementation of an ECC over the field $GF(p)$ (where p is prime) on a smart card. In the past, much research has focused on curves over the Galois field with 2^m elements, $GF(2^m)$, because it is possible to create efficient hardware implementations [6]. However, because of the speed advantages of elliptic curves over the field $GF(p)$ compared to $GF(2^m)$ when a crypto coprocessor for modular arithmetic is available [8], and because of patent issues associated with curves over $GF(2^m)$, this research has investigated curves over $GF(p)$.

The smart card targeted for the project is the Motorola M-Smart Jupiter™ smart card [15] based on Java Card™ 2.1 technology and an ARM processor [1] with a word size of 32 bits, 64 KB of ROM, 32KB of EEPROM, 3KB RAM and a modular arithmetic accelerator. All of the ECC operations were implemented in the C programming language, and testing was performed on a simulation of the smart card utilizing the ARM Software Development Toolkit.

We include algorithms for point addition and doubling in various coordinate systems and give details of the various speed, RAM usage, parameter and code size tradeoffs that are possible. Also included are the relative times for ECDSA [10] and RSA [13, pp.285-291] signature and verification operations of equivalent security. All PC timings given in the paper were performed on a Pentium III 450 MHz.

2 Field arithmetic

In order to achieve an efficient implementation of an ECC, it is crucial to have an efficient implementation of the underlying field arithmetic, which in this case is field arithmetic for $GF(p)$. The field operations of modular addition and subtraction are relatively fast and easily implemented. However, modular multiplication (which requires a modular reduction) and modular inversion are much more time consuming. Various methods of either speeding up or avoiding these operations have been published. These are discussed in the following subsections.

2.1 Selection of the modular reduction algorithm

Two efficient methods of modular reduction that are often considered for implementation and may be used with any modulus are Barrett reduction and Montgomery reduction [4] [13, pp.599–604]. Each of these methods requires a precomputation that depends on the modulus. The efficiency of both methods is due to the fact that the only divisions performed can be implemented as right shifts which are quite fast. However, Montgomery reduction also requires the operands to be converted to a special Montgomery form. If the precomputation and conversion time is ignored, Montgomery reduction is slightly faster than Barrett reduction, and both are faster than the classical algorithm [13, p.600].

Another modular reduction method which has previously been successfully adopted in a software only implementation by Brown et al. in [5] in order to increase the speed of the ECC is to use a modulus with a special form, such as the NIST primes [17], enabling a very fast but specialized reduction algorithm. In fact, Brown et al. achieved reduction timings that were between 6% and 33% of the time required for Barrett reduction, depending on the prime used and whether assembly language was used. ECCs using the NIST primes were considered in this research, but they did not give favourable timings because the coprocessor could not be effectively utilized in such an implementation. For example, for a 224 bit modulus, multiplication without reduction in software (which is necessary before the reduction takes place) took 4.9 times as long as a hardware modular multiplication. Also, the 224 bit modular reduction in software took 1.5 times as long as a hardware modular multiplication.

A pseudo-Mersenne prime [2] can also be used to speed up the reduction algorithm. A fast reduction algorithm for primes of this form is given in [13, p.605]. However, as for the NIST primes, in order to perform a modular multiplication, a multiplication without reduction is first required which takes much longer than a hardware modular multiplication on the smart card. Because the algorithms for special primes did not give any speed advantages on the smart card, the coprocessor was used to perform the modular arithmetic and random primes were used to define the elliptic curves that were used.

2.2 Modular inversion

Finding multiplicative inverses in the field $GF(p)$ (required by ECCs over $GF(p)$) is extremely slow (taking about 40 to 65 times as long as Barrett reduction [5]) and is generally avoided as much as possible. The use of coordinate systems other than the Affine coordinate system greatly reduces the number of inversions required in the operations of the ECC (see Section 3). However, an efficient inversion algorithm is still needed for those times when inversion is required such as during the creation of a DSA digital signature by one party, verification of the validity of that signature by another party or at the end of a scalar multiplication to convert the coordinates back to Affine coordinates. Three inversion algorithms were considered for use in this project, the binary extended GCD (BEGCD) algorithm [13, pp.608–610], the extended Euclidean algorithm (EEA) [13, p.67] and the exponentiation method (from Fermat's (little) theorem [13, p.69]), $a^{-1} \pmod{p} \equiv a^{p-2} \pmod{p}$. The EEA involves multi-precision divisions, which are quite slow. In order to avoid such divisions, the BEGCD algorithm uses right shifts (which are fast), but requires more iterations. The speed in software of both these methods was estimated for the smart card and compared to that of the exponentiation method. Because the exponentiation method was available in hardware, it required minimal code space and did not decrease performance compared to the EEA and BEGCD algorithms which were only available in software. For these reasons, the exponentiation method was chosen as the inversion algorithm.

3 Point coordinates

One of the crucial decisions when implementing an efficient elliptic curve cryptosystem over $GF(p)$ is deciding which point coordinate system to use. The point coordinate system used for addition and doubling of points on the elliptic curve determines the efficiency of these routines, and hence the efficiency of the basic cryptographic operation, scalar multiplication. This section analyses the efficiencies of the different coordinate systems considered. These coordinate systems were taken from [7] and are Affine (where a point is represented as $(x_\mathrm{A}, y_\mathrm{A})$ as in Section 1), Projective (represented as (X, Y, Z) where $x_\mathrm{A} = XZ^{-1}$ and $y_\mathrm{A} = YZ^{-1}$), and Jacobian, Modified Jacobian and Chudnovsky Jacobian (represented as (X, Y, Z), (X, Y, Z, aZ^4) and (X, Y, Z, Z^2, Z^3) respectively where $x_\mathrm{A} = XZ^{-2}$ and $y_\mathrm{A} = YZ^{-3}$). Note that [7] does not give detailed algorithms, and considerable effort has been spent minimizing the number of temporary variables required by each algorithm. Detailed addition and doubling algorithms that have been optimized to reduce the number of temporary variables for Jacobian, Chudnovsky Jacobian and Modified Jacobian coordinates are given in Appendix B, and descriptions and formulae for operations using these coordinate systems in [7] and [3].

Affine coordinates are the simplest to understand and are used for communication between two parties because they require the lowest bandwidth. However, the modular inversions required when

Table 1: Point conversion complexity: M is squaring or multiplication; and I is inversion.

From \ To	Affine	Projective	Jacobian	Chudnovsky	Modified
Affine	-	-	-	-	-
Projective	$2M + I$	-	$2M + I$	$2M + I$	$2M + I$
Jacobian	$4M + I$	$4M + I$	-	$2M$	$3M$
Chudnovsky	$4M + I$	$4M + I$	-	-	$3M$
Modified	$4M + I$	$4M + I$	-	$2M$	-

adding and doubling points which are represented using Affine coordinates cause them to be highly inefficient for use in addition and doubling of points. The other coordinate systems require at least one extra value to represent a point and do not require the use of modular inversions in point addition and doubling, but extra multiplications and squarings are required instead.

Cohen et al. [7] recommended the idea of mixed coordinates, where the inputs and outputs to point additions and doublings may be in different coordinates. This can be very efficient when scalar multiplication is implemented with the base point stored in Affine coordinates.

In order to use mixed coordinates it is sometimes necessary to convert a point representation from one coordinate system to another to have the input in the required format for the addition or doubling algorithm. Table 1 shows that conversion from Affine coordinates to any of the other coordinate systems is very efficient because the conversions only consist of setting all of the Z, Z^2 and Z^3 coordinates to one and the aZ^4 coordinate to a (the elliptic curve parameter). Conversion to or from Projective coordinates is inefficient because of the inversion required, as is converting from any of the other coordinate systems to Affine coordinates. However, conversions among the three Jacobian variants are quite efficient, and these are therefore used in mixed coordinate scalar multiplication.

Table 2 contains the times for addition and doubling in various coordinate systems. All calculations were performed for curves over a 160 bit prime. The first column specifies the coordinates used in the algorithm. For addition, the first two letters indicate the coordinates of the two input points. The third letter indicates the output coordinates. For example, AJM is an addition algorithm with input points in Affine and Jacobian coordinates and an output point in Modified Jacobian coordinates. For doubling, the first letter indicates the input point coordinates and the second letter indicates the output point coordinates. For example, MJ is a doubling algorithm with an input point in Modified Jacobian coordinates and an output point in Jacobian coordinates. Because three different Jacobian addition algorithms have been used (see Section 4 and Appendix B), these are distinguished with a number at the end of the acronym. The two different Jacobian doubling algorithms are distinguished in the same way.

See that when the actual Pentium timings and the smart card estimates are sorted according to speed, they are mostly in the same order, indicating that the estimations are reasonable. The table also gives the number of 160 bit variables that are required for each algorithm. Lastly, it gives times for converting Jacobian, Modified Jacobian or Chudnovsky Jacobian points into either Chudnovsky Jacobian or Modified Jacobian points.

Although the AAC, AAM, AJ and AM operations are very fast, these methods are not very useful because the output of an addition or doubling (used as input to these procedures) must be converted to Affine coordinates which is computationally intensive.

Table 2: Addition and Doubling Efficiencies

Acronym	Add	Subtract	Multiply	Square	Other	Mul. + Square	Pentium Timings (ms)	Smart card Estimate	Number of Variables
Point Addition									
AAC	1	6	4	2	0	6	0.027	46%	UC
AAM	1	6	5	3	0	8	0.035	54%	UC
AJJ2	0	7	8	3	0	11	-	68.05%	8
APP	1	6	9	2	0	11	-	68.17%	UC
AJJ1	1	6	8	3	0	11	-	68.17%	8
ACC	0	7	8	3	0	11	-	68.84%	8
AJM−3	2	8	8	5	0	13	-	77.67%	8
AMM−3	2	8	8	5	0	13	-	77.67%	8
AJM	0	7	9	5	0	14	-	78.62%	9*
AMM	0	7	9	5	0	14	-	78.62%	9*
AJJ3	4.5	6	8	3	Shift	11	-	83.50%	7
CCC	0	7	11	3	0	14	0.059	84.48%	11
PPP	1	6	12	2	0	14	0.059	86.93%	UC
JJJ1	1	6	12	4	0	16	0.067	91.90%	9
JJM	0	7	13	6	0	19	0.077	100.00%	10*
MMM	0	7	13	6	0	19	0.078	100.00%	11*
AAA	0	6	2	1	Inv.	3	0.237	640.40%	UC
Point Doubling									
AJ	5	3	5	2	0	7	0.030	UC	UC
MJ	8	4	3	4	0	7	0.033	50.03%	6
MM	9	4	4	4	0	8	0.038	58.00%	6
JJ2−3	8	5	4	4	0	8	-	58.67%	6
JJ1−3	5	5	4	4	Shift	8	-	59.75%	5
AM	5	3	5	4	0	9	0.038	UC	UC
CC	9	4	5	6	0	11	0.049	65.84%	6*
JJ1	5	4	4	6	Shift	10	0.044	67.91%	6*
PP	14	3	7	5	0	12	0.055	70.13%	UC
AA	4	4	2	2	Inv.	4	0.240	653.71%	UC
Point Conversion									
J,M to C	0	0	1	1	0	2	0.008	8.87%	
J,C to M	0	0	1	2	0	3	0.011	12.51%	

A	Affine	−3	Optimized for $a = p - 3$ [3, pp.59-60]
P	Projective	*	Including the a parameter
J	Jacobian	1, 2 or 3	Different versions for the same coordinate system
C	Chudnovsky Jacobian		
M	Modified Jacobian	UC	Uncalculated because inefficient

4 Scalar multiplication

Scalar multiplication is the basic cryptographic operation of an ECC, and consists of a series of point additions and doublings. The scalar multiplication algorithm chosen for the smart card implementation was the binary method [3, p.63], because it does not require a precomputation and therefore uses less memory, unlike other more efficient methods. One option when implementing the binary method is to use a signed representation of the scalar such as the non-adjacent form (NAF) [3, pp.67–68]. Because the NAF represents a scalar with a smaller number of non-zero digits, a lower number of point additions is required in a binary scalar multiplication using this representation since each non-zero digit corresponds to one point addition. The estimated scalar multiplication figures in Table 3 indicate that using a NAF scalar should make the scalar multiplication about 10% faster than when using an unsigned scalar. The values in Table 5 show an increase in efficiency of about 6% for the time required for a person to digitally sign a value and 4% to 17% (depending on the settings used) for the time required for another person to verify the validity of the digital signature.

Another option is to use the two-in-one variant of the binary algorithm that computes $k_1P + k_2Q$, where k_1 and k_2 are scalars and P and Q are points on the curve [13, p. 618], [18]. If there is insufficient memory to store $P + Q$ or $P - Q$, these points need not be stored, but may be computed each time they are needed. However, this can cause the algorithm to be slower, depending on the coordinates in which the temporary points $P+Q$ and $P-Q$ are stored and the time taken to convert the points to these coordinates. We estimate that a two-in-one scalar multiplication takes about 60% to 70% of the time that two separate scalar multiplications take, depending on the options chosen. The data in Table 5 indicates that DSA verification using this method actually takes 65% to 70% of the

Table 3: Estimated time for signed (naf) and unsigned scalar multiplication on the smart card using the Binary method

Addition Algorithm	Doubling Algorithm	Naf $a = p - 3$	Naf $a \neq p - 3$	Unsigned $a = p - 3$	Unsigned $a \neq p - 3$
AJM-3	MJ/MM	76.56%		87.22%	
AJJ2	JJ2-3	76.68%		87.08%	
AJM	MJ/MM		76.85%		87.67%
AJJ1	JJ1-3	77.74%		88.15%	
AMM-3	MM	79.03%		90.95%	
AMM	MM		79.33%		91.40%
AJJ1	MM		79.96%		92.36%
ACC	JJ1-3	80.70%		92.61%	
AJJ3	JJ1-3	82.49%		95.32%	
ACC	MM		82.92%		96.82%
JJM	MJ/MM		83.48%		97.67%
ACC	CC		83.72%		94.20%
AMM-3	JJ1-3	84.56%		98.45%	
AJJ1	JJ1		85.48%		95.84%
JJJ	MM		87.32%		101.40%
AJJ1	CC		86.26%		98.03%
JJJ	MM		87.32%		103.46%
APP	PP		87.57%		97.92%
CCC	MM		87.77%		104.14%
ACC	JJ1		88.43%		100.30%
CCC	CC		88.57%		101.52%
AJJ3	JJ1		90.23%		103.01%
AMM	JJ1		92.59%		106.58%
JJJ	JJ1		92.83%		106.94%
CCC	JJ1		93.28%		107.62%
AMM	CC		93.37%		108.77%
PPP	PP		93.39%		106.70%
JJJ	CC		93.61%		109.13%
MMM	JJ1		99.22%		116.58%
MMM	CC		100.00%		118.78%

time taken when not using the two-in-one scalar multiplication.

The basic coordinate system chosen for the smart card implementation was the mixed Jacobian and Modified Jacobian coordinate system, with one input to the addition in Affine coordinates (AJM/MJ/MM). This coordinate system was chosen because the estimates in Table 3 indicate that it is the most efficient coordinate system to use if $a \neq p-3$. One of the inputs to the addition was chosen to be Affine because of the faster implementation available and because fewer variables were required in this case.

In order to see how much efficiency could be gained by setting $a = p-3$, the AJM addition was modified slightly to create the AJM-3/MJ/MM coordinates. Because Jacobian coordinates allow a further speedup from setting $a = p-3$ which is not available when using Modified Jacobian coordinates, the AJM-3/MJ/MM algorithms were further modified to allow this speedup and to use only Jacobian coordinates, resulting in the AJJ2/JJ2-3 coordinates.

Because of the limited amount of memory available, Jacobian coordinates were also implemented in order to see how much speed needed to be sacrificed in order to use fewer variables. Two different addition algorithms were available—one that used three temporary variables but was faster (AJJ1), and one that used two temporary variables but was slower (AJJ3). These algorithms were implemented for $a \neq p-3$ and also optimized for $a = p-3$, giving the four sets of coordinates AJJ1/JJ1, AJJ1/JJ1-3, AJJ3/JJ1 and AJJ3/JJ1-3. Figure 1 shows the number of variables that are saved for each coordinate system and scalar multiplication setting.

Figure 2 displays the running time (as a percentage of the longest running time) of the ECDSA signature and verification for each of the options that was implemented. The settings used were signed or unsigned scalars (NAF or no NAF), two separate multiplications or

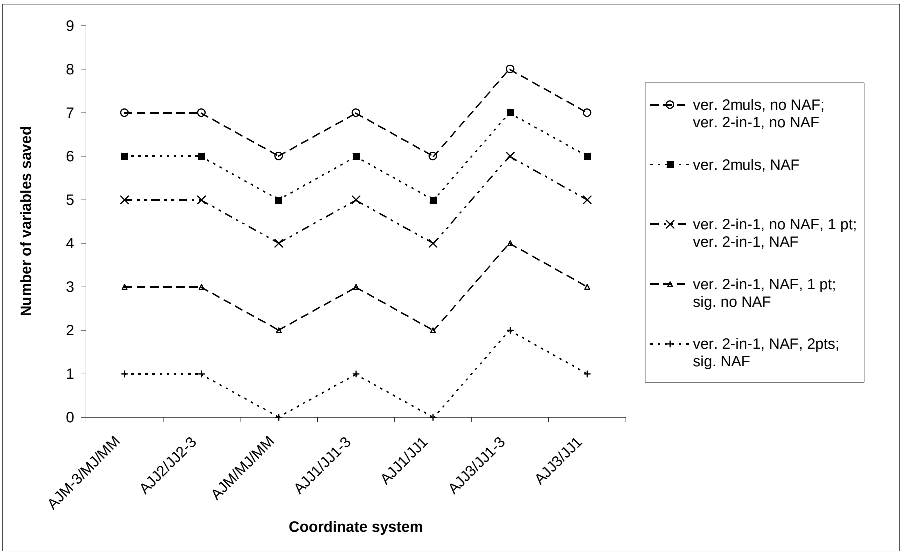

Figure 1: Number of variables saved for the different options for ECDSA

a two-in-one multiplication for the verification, and when a two-in-one multiplication was performed, whether there were two temporary points calculated at the beginning of the scalar multiplication ($P+Q$ and $P-Q$), one point ($P+Q$ for no NAF and $P-Q$ for NAF) or no points. Note that the temporary points were stored in Affine format in order to be able to guarantee one Affine input to the addition algorithm; however, the time to calculate the points may outweigh any time saved. Although storing the points in Jacobian coordinates may give a faster implementation by avoiding the inversion per point needed to convert them to Affine, this option was not implemented because of the increased code size for the addition and increased number of variables required. In any case, the time taken to calculate each point is only about 2% of the total verification time (this is 1% on the graph where 100% is the slowest verification speed), and thus storing the points in Jacobian format would not greatly increase efficiency, bearing in mind that even less than the 2% of verification time per point would be saved because of the slower addition algorithm being used.

Figure 2 shows that the AJM/MJ/MM coordinates are best when $a \neq p-3$. When $a = p-3$ and the NAF form of the scalar is used, the AJM-3/MJ/MM coordinates are fastest. Using a NAF scalar always gives a faster result than an unsigned scalar and the two-in-one multiplication algorithm enables a faster verification.

Figure 3 gives the code size for each of the different implementations. Because the interface assumes that all points are passed to it in compressed form (that is, an Affine x-coordinate and one byte to specify the sign of the y-coordinate and point format), a point uncompression procedure was implemented. The main part of this procedure is the square root algorithm, which is long when $p \equiv 1 \pmod 4$. It is possible to save a further 528 bytes of code space from any of the implementations by omitting the point uncompres-

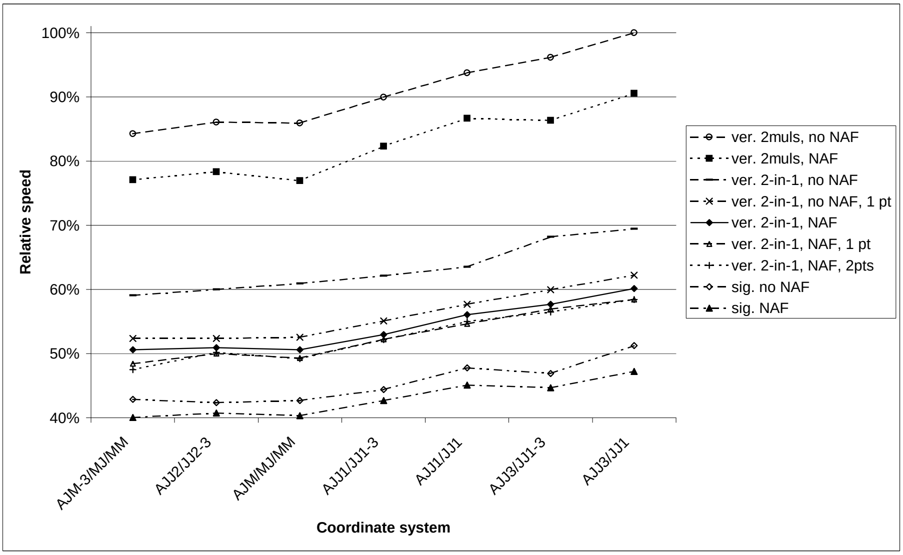

Figure 2: Ecdsa signature and verification timings on the smart card simulator

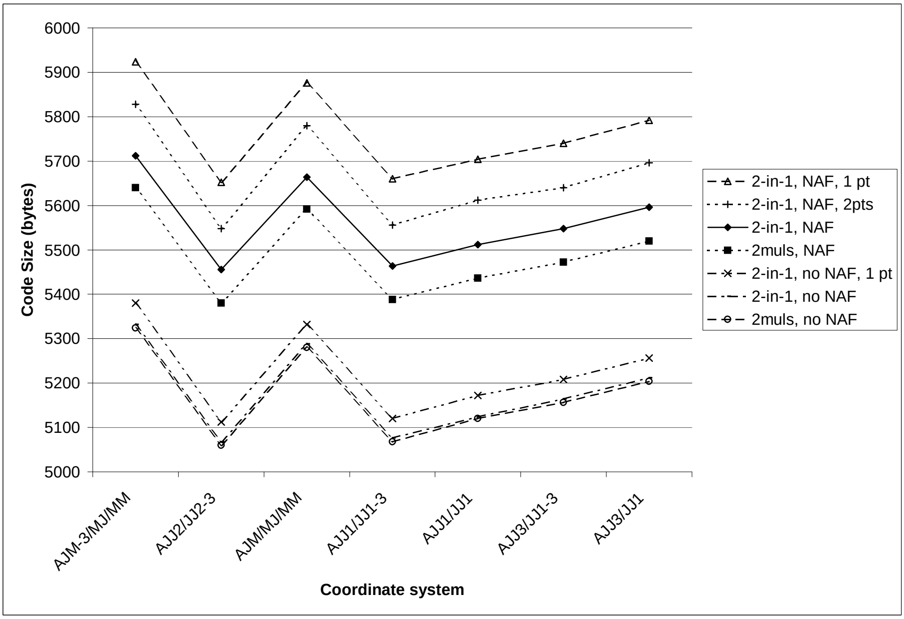

Figure 3: Code size for ECDSA signature and verification for the smart card.

sion procedure for $p \equiv 1 \pmod 4$ and not using these curves.

The optimal choice of coordinate system and scalar multiplication algorithm depends on the importance of speed compared to code size and minimal RAM usage. If speed is considered the most important, the best compromise may be to choose a signed scalar with two-in-one multiplication and no temporary points stored and using either the AJM/MJ/MM or AJJ2/JJ2-3 coordinates, whichever is appropriate. This gives good signature and verification speeds, and saves a medium amount of code space and variables.

5 Comparison of RSA with ECDSA

Table 4 compares the speed of ECDSA (Elliptic Curve Digital Signature Algorithm) signing and verifying (using the two-in-one scalar multiplication algorithm with two precomputed points, a signed scalar and AJM/MJ/MM coordinates) to the speed of RSA signing and verifying on the smart card and the publicly available MIRACL library [16] on a Pentium. It should be noted that the RSA verification times are faster because a small exponent has been used. The table gives the times as a percentage of the EC signature time on that platform. These results demonstrate that the ratio of the time taken for the EC operations to the time taken for the RSA signature is about the same for both the smart card and the Pentium. However, on the smart card, the RSA signature was mostly performed in hardware, whereas a large number of the EC computations had to be performed in software, which would slow down the EC. That the RSA signature time on the smart card is longer than the time for the Pentium shows that we have quite an efficient implementation of ECDSA verification and signing on the smart card.

Table 4: Ecdsa and Rsa Time Comparison on Smart Card and Pentium

Algorithm		Miracl Library Pentium III 450 MHz	Our library Smart Card
160 bit ECDSA	Signature	100%	100%
	Verification	121%	122%
1024 bit RSA	Signature	220%	236%
	Verification	29%	24%

6 Conclusion

We have investigated the efficiency of various coordinate systems and scalar multiplication algorithms available when implementing an elliptic curve cryptosystem over the field $GF(p)$ on a smart card with a coprocessor for support of modular arithmetic operations. Several coordinate systems have been implemented and timed, as well as scalar multiplication algorithms using signed and unsigned scalars to find k_1P and $k_1P + k_2Q$ (where k_1 and k_2 are scalars and P and Q are points on the curve). The code size and the number of variables (where each variable is the same size as the modulus) required for each different implementation have also been investigated. A fast coordinate system and scalar multiplication algorithm with medium code size and variable usage have been recommended. The data in this paper is also sufficient to make an informed choice of algorithm for other requirements. Algorithms for addition and doubling in Jacobian, Chudnovsky Jacobian and Modified Jacobian coordinates with a minimum number of temporary variables are presented in Appendix B, and the choice of the prime p and modular reduction and inversion algorithms discussed.

Acknowledgement: this research is part of an ARC SPIRT project (C10024103) undertaken jointly by Queensland University of Technology and Motorola.

References

[1] Atmel Corporation. Motorola chooses Atmel technology for its M-Smart Jupiter smart card platform, Press release. http://www.armltd.co.uk/sitearchitek/news.ns4/916d9b71de20659380256944003588dec/cd91a1cbf6c291228025692f002a0018!OpenDocument (accessed 09/08/2001), 23/06/1999. C358

[2] Daniel V. Bailey and Christof Paar. Optimal extension fields for fast arithmetic in public-key algorithms. In *Advances in Cryptology—Crypto '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 472–485. Springer-Verlag, 1998. C359

[3] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*, volume 265 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 1999. C357, C361, C365

[4] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Comparison of three modular reduction functions. In *Advances in Cryptology—Crypto '93*, volume 773 of *Lecture Notes in Computer Science*, pages 175–186. Springer-Verlag, 1994. C359

[5] M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the NIST elliptic curves over prime fields.

In *Topics in Cryptology—CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 250–265. Springer-Verlag, 2001. http://www.cacr.math.uwaterloo.ca/~ajmeneze/research.html (accessed 23/01/2001). C359, C360

[6] Certicom Corp. The elliptic curve cryptosystem for smart cards, The seventh in a series of ECC white papers. http://www.certicom.com/research.html (accessed 04/05/2000), May, 1998. C357

[7] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *Advances in Cryptology—ASIACRYPT '98, Proceedings*, volume 1514 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 1998. C356, C361, C362, C377

[8] Erik De Win, Serge Mister, Bart Preneel, and Michael Wiener. On the performance of signature schemes based on elliptic curves. In *Algorithmic Number Theory: Third International Symposium, ANTS-III, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 1998. C357

[9] Toshio Hasegawa, Junko Nakajima, and Mitsuru Matsui. A practical implementation of elliptic curve cryptosystems over $GF(p)$ on a 16-bit microcomputer. In *Public Key Cryptography – PKC '98,Proceedings*, volume 1431 of *Lecture Notes in Computer Science*, pages 182–194. Springer-Verlag, 1998. C356, C377

[10] IEEE. IEEE Std 1363-2000, IEEE standard specifications for public-key cryptography. http://ieeexplore.ieee.org/login.html (accessed 15/05/2001), 2000. C358

[11] Neil Koblitz. Elliptic curve cryptosystems. In *Mathematics of Computation*, volume 48, pages 203–209, 1987. C356

[12] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, 1986. C356

[13] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. C358, C359, C360, C365

[14] Victor S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology—Proceedings of Crypto 85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag, 1986. C356

[15] Motorola, Inc. Motorola ships industry's first 32-bit RISC Java Card 2.1^{TM} technology/Visa Open Platform 2.0 card, Press release. http://www.mot.com/LMPS/pressreleases/page483.htm (accessed 08/08/2001), 28/03/2000. C358

[16] Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL). http://indigo.ie/~mscott/ (accessed 23/06/2000). C372

[17] National Institute of Standards and Technology. Digital signature standard (DSS). Technical report, FIPS Publication 186-2, January 2000. http://www.csrc.nist.gov/publications/fips/ (accessed 07/06/2001). C359

[18] Huapeng Wu, M. Anwarul Hasan, and Ian F. Blake. Efficient computation of multiple points for elliptic curve cryptosystems. In *1998 IEEE International Symposium on Information Theory, Proceedings*, page 49, 1998. C365

A ECDSA smart card simulation data

Table 5 displays the ECDSA verification and signature timings, code size and number of variables *saved* (not used) for each different efficiency option. This data is also displayed graphically in Figures 1, 2 and 3.

B Point addition and doubling algorithms

Tables 6, 7 and 8 give the algorithms for addition and doubling in Jacobian coordinates and variants. The JJJ3, AJJ3, JJ1 and JJ1-3 algorithms have been taken from [9]. However, checks have been added to ensure that the point at infinity is not one of the points being added and that the points being added are not identical or each other's negative.

The other algorithms are derived from the formulae given in [7]. The two different Jacobian addition algorithms have been included because JJJ1 and AJJ1 are faster, but JJJ3 and AJJ3 require one less variable. Each algorithm assumes that the output will overwrite an input point.

Table 5: Code size, number of variables saved and timings for the smart card

2-in-1 Multiplication	Number of Temporary Points	Signed (NAF) Scalar Used	Addition Algorithm	Doubling Algorithm(s)	ECDSA Signature Time (Simulated)	ECDSA Verification Time (Simulated)	Code Size (bytes)	Variables Saved in Signature	Variables Saved in Verification
no	0	no	AJM-3	MJ/MM	42.9%	84.3%	5324	3	7
no	0	no	AJJ2	JJ2-3	42.3%	86.1%	5060	3	7
no	0	no	AJM	MJ/MM	42.7%	85.9%	5280	2	6
no	0	no	AJJ1	JJ1-3	44.4%	90.0%	5068	3	7
no	0	no	AJJ1	JJ1	47.7%	93.7%	5120	2	6
no	0	no	AJJ3	JJ1-3	46.9%	96.1%	5156	4	8
no	0	no	AJJ3	JJ1	51.2%	100.0%	5204	3	7
no	0	yes	AJM-3	MJ/MM	40.1%	77.1%	5640	1	6
no	0	yes	AJJ2	JJ2-3	40.7%	78.3%	5380	1	6
no	0	yes	AJM	MJ/MM	40.3%	76.9%	5592	0	5
no	0	yes	AJJ1	JJ1-3	42.6%	82.3%	5388	1	6
no	0	yes	AJJ1	JJ1	45.0%	86.6%	5436	0	5
no	0	yes	AJJ3	JJ1-3	44.6%	86.3%	5472	2	7
no	0	yes	AJJ3	JJ1	47.2%	90.5%	5520	1	6
yes	0	no	AJM-3	MJ/MM	42.9%	59.0%	5332	3	7
yes	0	no	AJJ2	JJ2-3	42.4%	60.0%	5068	3	7
yes	0	no	AJM	MJ/MM	42.7%	60.9%	5288	2	6
yes	0	no	AJJ1	JJ1-3	44.4%	62.1%	5076	3	7
yes	0	no	AJJ1	JJ1	47.8%	63.5%	5124	2	6
yes	0	no	AJJ3	JJ1-3	46.9%	68.2%	5164	4	8
yes	0	no	AJJ3	JJ1	51.3%	69.4%	5212	3	7
yes	0	yes	AJM-3	MJ/MM	40.2%	50.6%	5712	1	5
yes	0	yes	AJJ2	JJ2-3	40.8%	50.9%	5456	1	5
yes	0	yes	AJM	MJ/MM	40.2%	50.6%	5664	0	4
yes	0	yes	AJJ1	JJ1-3	42.7%	53.0%	5464	1	5
yes	0	yes	AJJ1	JJ1	45.1%	56.1%	5512	0	4
yes	0	yes	AJJ3	JJ1-3	44.7%	57.7%	5548	2	6
yes	0	yes	AJJ3	JJ1	47.3%	60.1%	5596	1	5
yes	1	no	AJM-3	MJ/MM	42.9%	52.3%	5380	3	5
yes	1	no	AJJ2	JJ2-3	42.4%	52.4%	5112	3	5
yes	1	no	AJM	MJ/MM	42.7%	52.5%	5332	2	4
yes	1	no	AJJ1	JJ1-3	44.4%	55.1%	5120	3	5
yes	1	no	AJJ1	JJ1	47.8%	57.7%	5172	2	4
yes	1	no	AJJ3	JJ1-3	46.9%	59.9%	5208	4	6
yes	1	no	AJJ3	JJ1	51.2%	62.2%	5256	3	5
yes	1	yes	AJM-3	MJ/MM	40.1%	49.4%	5924	1	3
yes	1	yes	AJJ2	JJ2-3	40.8%	50.0%	5652	1	3
yes	1	yes	AJM	MJ/MM	40.3%	49.3%	5876	0	2
yes	1	yes	AJJ1	JJ1-3	42.7%	52.2%	5660	1	3
yes	1	yes	AJJ1	JJ1	45.1%	54.6%	5704	0	2
yes	1	yes	AJJ3	JJ1-3	44.7%	56.9%	5740	2	4
yes	1	yes	AJJ3	JJ1	47.2%	58.4%	5792	1	3
yes	2	yes	AJM-3	MJ/MM	40.0%	49.5%	5828	1	1
yes	2	yes	AJJ2	JJ2-3	40.7%	50.2%	5548	1	1
yes	2	yes	AJM	MJ/MM	40.3%	49.2%	5780	0	0
yes	2	yes	AJJ1	JJ1-3	42.6%	52.2%	5556	1	1
yes	2	yes	AJJ1	JJ1	45.1%	55.0%	5612	0	0
yes	2	yes	AJJ3	JJ1-3	44.6%	56.5%	5640	2	2
yes	2	yes	AJJ3	JJ1	47.2%	58.4%	5696	1	1

Table 6: Jacobian 1 and 3 Addition and Jacobian 1 Doubling

JJJ1 and AJJ1 Addition $Q = Q + P$, where $Q = (X, Y, Z)$ and $P = (X_2, Y_2)$ or (X_2, Y_2, Z_2)	JJJ3 and AJJ3 Addition $Q = Q + P$, where $Q = (X, Y, Z)$ and $P = (X_2, Y_2)$ or (X_2, Y_2, Z_2)	JJ1 and JJ1 − 3 Doubling $Q = Q + Q$, where $Q = (X, Y, Z)$

JJJ1 and AJJ1 Addition:

```
if ($P == \phi$) return $Q$
if ($Z == 0$)
{   $Q = P$
    return $Q$   }
if ($P$ is not Affine and $Z_2 \neq 1$)
{   $T_1 = Z_2^2$
    $X = X * T_1$
    $T_1 = Z_2 * T_1$
    $Y = Y * T_1$   }
$T_1 = Z^2$
$T_2 = X_2 * T_1$
$T_1 = Z * T_1$
$T_1 = Y_2 * T_1$
$T_1 = T_1 - Y$
$T_2 = T_2 - X$
if ($T_2 == 0$)
{   if ($T_1 == 0$)
    {   $Q = P$
        Double ($Q$)
        return $Q$   }
    else
    {   $Z = 0$
        return $Q$   }   }
if ($P$ is not Affine and $Z_2 \neq 1$)
{   $Z = Z * Z_2$   }
$Z = Z * T_2$
$T_3 = T_2^2$
$T_2 = T_2 * T_3$
$T_3 = T_3 * X$
$X = T_1^2$
$Y = T_2 * Y$
$T_2 = X - T_2$
$X = 2T_3$
$X = T_2 - X$
$T_2 = T_3 - X$
$T_2 = T_1 * T_2$
$Y = T_2 - Y$
```

JJJ3 and AJJ3 Addition:

```
if ($P == \phi$) return $Q$
if $Z == 0$
{   $Q = P$
    return $Q$   }
if ($P$ is not Affine and $Z_2 \neq 1$)
{   $T_1 = Z_2^2$
    $X = X * T_1$
    $T_1 = Z_2 * T_1$
    $Y = Y * T_1$   }
$T_1 = Z^2$
$T_2 = X_2 * T_1$
$T_1 = Z * T_1$
$T_1 = Y_2 * T_1$
$Y = Y - T_1$
$T_1 = 2T_1$
$T_1 = Y + T_1$
$X = X - T_2$
if ($X == 0$)
{   if ($Y == 0$)
    {   $Q = P$
        Double ($Q$)
        return $Q$   }
    else
    {   $Z = 0$
        return $Q$   }   }
$T_2 = 2T_2$
$T_2 = X + T_2$
if ($P$ is not Affine and $Z_2 \neq 1$)
{   $Z = Z * Z_2$   }
$Z = Z * X$
$T_1 = T_1 * X$
$X = X^2$
$T_2 = T_2 * X$
$T_1 = T_1 * X$
$X = Y^2$
$X = X - T_2$
$T_2 = T_2 - X$
$T_2 = T_2 - X$
$T_2 = T_2 * Y$
$Y = T_2 - T_1$
$Y = Y/2$
```

JJ1 and JJ1 − 3 Doubling:

```
if ($Z == 0$) return $Q$
$T_1 = Z^2$
$Z = Y * Z$
$Z = 2Z$
if ($a == p - 3$)
{   $T_2 = X - T_1$
    $T_1 = X + T_1$
    $T_2 = T_1 * T_2$
    $T_1 = 2T_2$
    $T_1 = T_1 + T_2$   }
else
{   $T_1 = T_1^2$
    $T_1 = a * T_1$
    $T_2 = X^2$
    $T_1 = T_2 + T_1$
    $T_2 = 2T_2$
    $T_1 = T_2 + T_1$   }
$Y = 2Y$
$Y = Y^2$
$T_2 = Y^2$
$Y = Y * X$
$T_2 = T_2/2$
$X = T_1^2$
$X = X - Y$
$X = X - Y$
$Y = Y - X$
$Y = Y * T_1$
$Y = Y - T_2$
```

Table 7: Modified Jacobian and Variants Addition and Doubling

```
AJM, AJM-3, JJM, MMM, AJJ2,
   AMM and AMM-3 Addition
$Q_{out} = Q_{in} + P$, where
$Q_{in} = (X, Y, Z)$ or $(X, Y, Z, aZ^4)$
$P = (X_2, Y_2)$, $(X_2, Y_2, Z_2)$ or $(X_2, Y_2, Z_2, aZ_2^4)$
if AJJ2
{ $Q_{out} = (X, Y, Z)$ and
  $aZ^4$ below is a temporary variable }
else
{$Q_{out} = (X, Y, Z, aZ^4)$}
```

```
if $(P == \phi)$
{ $aZ^4 = a * Z^4$ if necessary
  return $Q_{out}$ }
if $(Z == 0)$
{ $Q_{out} = P$
  return $Q_{out}$ }
if $(P$ is not Affine and $Z_2 \neq 1)$
{ $aZ^4 = Z_2^2$
  $X = X * aZ^4$
  $aZ^4 = Z_2 * aZ^4$
  $Y = Y * aZ^4$ }
$aZ^4 = Z^2$
$T_1 = X_2 * aZ^4$
$T_1 = T_1 - X$
$aZ^4 = Z * aZ^4$
$aZ^4 = Y_2 * aZ^4$
$aZ^4 = aZ^4 - Y$
if $(P$ is not Affine and $Z_2 \neq 1)$
{ $Z = Z * Z_2$ }
$Z = Z * T_1$
if $(T_1 == 0)$
{ if $(aZ^4 == 0)$
    { $Q_{out} = P$
      Double $(Q_{out})$
      return $Q_{out}$ }
  else
    { $Z = 0$
      $aZ^4 = 0$
      return $Q_{out}$ } }
$T_2 = T_1^2$
$T_1 = T_1 * T_2$
$Y = T_1 * Y$
$T_2 = X * T_2$
$X = \left(aZ^4\right)^2$
$X = X - T_1$
$X = X - T_2$
$X = X - T_2$
$T_2 = T_2 - X$
$T_2 = aZ^4 * T_2$
$Y = T_2 - Y$
if not doing AJJ2
{ $aZ^4 = Z^2$
  $aZ^4 = \left(aZ^4\right)^2$
  if $a == p - 3$
  { $aZ^4 = 0 - 3aZ^4$ }
  else
  {$aZ^4 = a * aZ^4$} }
```

```
MM, MJ and $JJ2 - 3$
      Doubling

$Q_{out} = Q_{in} + Q_{in}$ where
$Q_{out} = (X, Y, Z)$ or $(X, Y, Z, aZ^4)$
if AJJ2
{$Q_{in} = (X, Y, Z)$}
else
{$Q_{in} = (X, Y, Z, aZ^4)$}
```

```
if $(Z == 0)$ return $Q_{out}$
if doing JJ2-3
{$T_2 = Z^2$}
$T_1 = 2Y$
$Z = T_1 * Z$
$Y = Y^2$
$T_1 = 2X$
$T_1 = 2T_1$
$T_1 = T_1 * Y$
if doing JJ2-3
{ $T_2 = (X - T_2) * (X + T_2)$ #
  $X = 2T_2$
  $T_2 = T_2 + X$ }
else
{ $T_2 = X^2$
  $X = 2T_2$
  $T_2 = X + T_2$
  $T_2 = T_2 + aZ^4$ }
$X = T_2^2$
$X = X - T_1$
$X = X - T_1$
$T_1 = T_1 - X$
$T_2 = T_2 * T_1$
$Y = 2Y$
$Y = Y^2$
$Y = 2Y$
if doing MM
{ $T_1 = 2Y$
  $aZ^4 = T_1 * aZ^4$ }
$Y = T_2 - Y$
```

# This line may be calculated as:

```
$T_3 = X - T_2$
$X = X + T_2$
$T_2 = X * T_3$
```

In the smart card implementation, it was possible to use a coprocessor register in place of T_3.
By using an extra addition, it is also possible to compute T_2 without using the additional variable:

```
$X = X - T_2$
$T_2 = T_2 + T_2$
$T_2 = X + T_2$
$T_2 = T_2 * X$
```

TABLE 8: Chudnovsky Jacobian Addition and Doubling

ACC and CCC Addition
$Q = Q + P$, where
$Q = (X, Y, Z, Z^2, Z^3)$
$P = (X_2, Y_2)$ or $(X_2, Y_2, Z_2, Z_2^2, Z_2^3)$

```
if $(P == \phi)$
{ return $Q$ }
if $(Z == 0)$
{
    $Q = P$
    return $Q$
}
if $(P$ is not Affine and $Z_2 \neq 1)$
{ $X = X * \left(Z_2^2\right)$ }
$Z^2 = X_2 * \left(Z^2\right)$
$Z^2 = \left(Z^2\right) - X$
$T_1 = \left(Z^2\right)^2$
if $(P$ is not Affine and $Z_2 \neq 1)$
{ $Z = Z * Z_2$ }
$Z = \left(Z^2\right) * Z$
$Z^2 = \left(Z^2\right) * T_1$
$T_1 = X * T_1$
if $(P$ is not Affine and $Z_2 \neq 1)$
{ $Y = Y * \left(Z_2^3\right)$ }
$Z^3 = Y_2 * \left(Z^3\right)$
$Z^3 = \left(Z^3\right) - Y$
if $\left(\left(Z^2\right) == 0\right)$
{
    if $\left(\left(Z^3\right) == 0\right)$
    {
        $Q = P$
        Double $(Q)$
        return $Q$
    }
    else
    {
        $Z = 0$
        $Z^2 = 0$
        $Z^3 = 0$
        return $Q$
    }
}
$X = \left(Z^3\right)^2$
$X = X - \left(Z^2\right)$
$X = X - T_1$
$X = X - T_1$
$T_1 = T_1 - X$
$Z^3 = \left(Z^3\right) * T_1$
$Y = Y * \left(Z^2\right)$
$Y = \left(Z^3\right) - Y$
$Z^2 = (Z)^2$
$Z^3 = Z * \left(Z^2\right)$
```

CC Doubling
$Q = Q + Q$, where
$Q = (X, Y, Z, Z^2, Z^3)$

```
if $(Z == 0)$ return $Q$
$Z = Y * Z$
$Z = 2Z$
$Y = Y^2$
$Z^3 = X * Y$
$Z^3 = 2\left(Z^3\right)$
$Z^3 = 2\left(Z^3\right)$
$X = X^2$
$Z^2 = \left(Z^2\right)^2$
$Z^2 = a * \left(Z^2\right)$
$Z^2 = \left(Z^2\right) + X$
$Z^2 = \left(Z^2\right) + X$
$Z^2 = \left(Z^2\right) + X$
$X = \left(Z^2\right)^2$
$X = X - \left(Z^3\right)$
$X = X - \left(Z^3\right)$
$Z^3 = \left(Z^3\right) - X$
$Z^3 = \left(Z^3\right) * \left(Z^2\right)$
$Y = Y^2$
$Y = 2Y$
$Y = 2Y$
$Y = 2Y$
$Y = \left(Z^3\right) - Y$
$Z^2 = (Z)^2$
$Z^3 = Z * \left(Z^2\right)$
```