Standards for Efficient Cryptography

SEC 1: Elliptic Curve Cryptography

Certicom Research
Contact: Daniel R. L. Brown (dbrown@certicom.com)

May 21, 2009
Version 2.0

Contents

List of Tables

List of Figures

1 Introduction

This section gives an overview of this standard, its use, its aims, and its development.

1.1 Overview

This document specifies public-key cryptographic schemes based on elliptic curve cryptography (ECC). In particular, it specifies:

- signature schemes;
- encryption and key transport schemes; and
- key agreement schemes.

It also describes cryptographic primitives which are used to construct the schemes, and ASN.1 syntax for identifying the schemes.

The schemes are intended for general application within computer and communications systems.

1.2 Aim

The aim of this document is threefold:

- Firstly, to facilitate deployment of ECC by completely specifying efficient, well-established, and well-understood public-key cryptographic schemes based on ECC.
- Secondly, to encourage deployment of interoperable implementations of ECC by profiling standards such as ANS X9.62 [X9.62a], WAP WTLS [WTLS], ANS X9.63 [X9.63] and IEEE 1363 [1363], and recommendation NIST SP 800-56 [800-56A], but restricting the options allowed in these standards to increase the likelihood of interoperability and to ensure conformance with as many standards as possible.
- Thirdly, to help ensure ongoing detailed analysis of ECC by cryptographers by clearly, completely, and publicly specifying baseline techniques.

1.3 Compliance

Implementations may claim compliance with the cryptographic schemes specified in this document provide the external interface (input and output) to the schemes is equivalent to the interface specified here. Internal computations may be performed as specified here, or may be performed via an equivalent sequence of operations.

Note that this compliance definition implies that conformant implementations must perform all the cryptographic checks included in the scheme specifications in this document. This is important because the checks are essential for the prevention of subtle attacks.

It is intended that a validation system will be made available so that implementers can check compliance with this document — see the SECG website, `http://www.secg.org`, for further information.

1.4 Document Evolution

This document will be reviewed every five years to ensure it remains up to date with cryptographic advances.

This document is version 2.0.

Additional intermittent reviews may also be performed occasionally, as deemed necessary by the Standards for Efficiency Cryptography Group.

1.5 Intellectual Property

The reader's attention is called to the possibility that compliance with this document may require use of an invention covered by patent rights. By publication of this document, no position is taken with respect to the validity of this claim or of any patent rights in connection therewith. The patent holder(s) may have filed with the SECG a statement of willingness to grant a license under these rights on reasonable and nondiscriminatory terms and conditions to applicants desiring to obtain such a license. Additional details may be obtained from the patent holder and from the SECG website, `http://www.secg.org`.

1.6 Organization

This document is organized as follows.

The main body of the document focuses on the specification of public-key cryptographic schemes based on ECC. Section 2 describes the mathematical foundations fundamental to the operation of all the schemes. Section 3 provides the cryptographic components used to build the schemes. Sections 4, 5, and 6 respectively specify signature schemes, encryption and key transport schemes, and key agreement schemes.

The appendices to the document provide additional relevant material. Appendix A gives a glossary of the acronyms and notation used, as well as an explanation of the terms used. Appendix B elaborates some of the details of the main body — discussing implementation guidelines, making security remarks, and attributing references. Appendix C provides reference ASN.1 syntax for implementations to use to identify the schemes, and Appendix D lists the references cited in the document.

2 Mathematical Foundations

This section gives an overview of the mathematical foundations necessary for elliptic curve cryptography.

Use of each of the public-key cryptographic schemes described in this document involves arithmetic operations on an elliptic curve over a finite field. This section introduces the mathematical concepts necessary to understand and implement these arithmetic operations.

Section 2.1 discusses finite fields, Section 2.2 discusses elliptic curves over finite fields, and Section 2.3 describes the data types involved and the conventions used to convert between data types.

See Appendix B for a commentary on the contents on this section, including implementation discussion, security discussion, and references.

2.1 Finite Fields

Abstractly, a finite field consists of a finite set of objects called field elements together with the description of two operations — addition and multiplication — that can be performed on pairs of field elements. These operations must possess certain properties.

It turns out that there is a finite field containing q field elements if and only if q is a power of a prime number, and furthermore that for each such q there is precisely one finite field. The finite field containing q elements is denoted by $\mathbb{F}_q$.

Here only two types of finite fields $\mathbb{F}_q$ are used — finite fields $\mathbb{F}_p$ with $q = p$ an odd prime which are called prime finite fields, and finite fields $\mathbb{F}_{2^m}$ with $q = 2^m$ for some $m \geq 1$ which are called characteristic 2 finite fields.

It is necessary to describe these fields concretely in order to precisely specify cryptographic schemes based on ECC. Section 2.1.1 describes prime finite fields and Section 2.1.2 describes characteristic 2 finite fields.

2.1.1 The Finite Field $\mathbb{F}_p$

The finite field $\mathbb{F}_p$ is the prime finite field containing p elements. Although there is only one prime finite field $\mathbb{F}_p$ for each odd prime p, there are many different ways to represent the elements of $\mathbb{F}_p$.

Here the elements of $\mathbb{F}_p$ should be represented by the set of integers:

$$\{0, 1, \ldots, p-1\}$$

with addition and multiplication defined as follows:

- Addition: If $a, b \in \mathbb{F}_p$, then $a + b = r$ in $\mathbb{F}_p$, where $r \in [0, p-1]$ is the remainder when the integer $a + b$ is divided by p. This is known as addition modulo p and written $a + b \equiv r \pmod{p}$.

- Multiplication: If $a, b \in \mathbb{F}_p$, then $ab = s$ in $\mathbb{F}_p$, where $s \in [0, p-1]$ is the remainder when the integer ab is divided by p. This is known as multiplication modulo p and written $ab \equiv s \pmod{p}$.

Addition and multiplication in $\mathbb{F}_p$ can be calculated efficiently using standard algorithms for ordinary integer arithmetic. In this representation of $\mathbb{F}_p$, the additive identity or zero element is the integer 0, and the multiplicative identity is the integer 1.

It is convenient to define subtraction and division of field elements just as it is convenient to define subtraction and division of integers. To do so, the additive inverse (or negative) and multiplicative inverse of a field element must be described:

- Additive inverse: If $a \in \mathbb{F}_p$, then the additive inverse $(-a)$ of a in $\mathbb{F}_p$ is the unique solution to the equation $a + x \equiv 0 \pmod{p}$.
- Multiplicative inverse: If $a \in \mathbb{F}_p$, $a \neq 0$, then the multiplicative inverse a^{-1} of a in $\mathbb{F}_p$ is the unique solution to the equation $ax \equiv 1 \pmod{p}$.

Additive inverses and multiplicative inverses in $\mathbb{F}_p$ can be calculated efficiently. Multiplicative inverses can be calculated using the extended Euclidean algorithm. Division and subtraction are defined in terms of additive and multiplicative inverses: $a - b \bmod p$ is $a + (-b) \bmod p$ and $a/b \bmod p$ is $a(b^{-1}) \bmod p$.

Here the prime finite fields $\mathbb{F}_p$ used should have:

$$\lceil \log_2 p \rceil \in \{192, 224, 256, 384, 521\}.$$

This restriction is designed to facilitate interoperability, while enabling implementers to deploy implementations which are efficient in terms of computation and communication since p is aligned with word size, and which are capable of furnishing all commonly required security levels. Inclusion of $\lceil \log_2 p \rceil = 521$ instead of $\lceil \log_2 p \rceil = 512$ is an anomaly chosen to align this document with other standards efforts — in particular with the U.S. government's recommended elliptic curve domain parameters [186-2].

2.1.2 The Finite Field $\mathbb{F}_{2^m}$

The finite field $\mathbb{F}_{2^m}$ is the characteristic 2 finite field containing 2^m elements. Although there is only one characteristic 2 finite field $\mathbb{F}_{2^m}$ for each power 2^m of 2 with $m \geq 1$, there are many different ways to represent the elements of $\mathbb{F}_{2^m}$.

Here the elements of $\mathbb{F}_{2^m}$ should be represented by the set of binary polynomials of degree $m-1$ or less:

$$\{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0 : a_i \in \{0, 1\}\}$$

with addition and multiplication defined in terms of an irreducible binary polynomial $f(x)$ of degree m, known as the reduction polynomial, as follows:

- Addition: If $a = a_{m-1}x^{m-1} + \cdots + a_0$, $b = b_{m-1}x^{m-1} + \cdots + b_0 \in \mathbb{F}_{2^m}$, then $a + b = r$ in $\mathbb{F}_{2^m}$, where $r = r_{m-1}x^{m-1} + \cdots + r_0$ with $r_i \equiv a_i + b_i \pmod 2$.
- Multiplication: If $a = a_{m-1}x^{m-1} + \cdots + a_0$, $b = b_{m-1}x^{m-1} + \cdots + b_0 \in \mathbb{F}_{2^m}$, then $ab = s$ in $\mathbb{F}_{2^m}$, where $s = s_{m-1}x^{m-1} + \cdots + s_0$ is the remainder when the polynomial ab is divided by $f(x)$ with all coefficient arithmetic performed modulo 2.

Addition and multiplication in $\mathbb{F}_{2^m}$ can be calculated efficiently using standard algorithms for ordinary integer and polynomial arithmetic. In this representation of $\mathbb{F}_{2^m}$, the additive identity or zero element is the polynomial 0, and the multiplicative identity is the polynomial 1.

Again it is convenient to define subtraction and division of field elements. To do so, the additive inverse (or negative) and multiplicative inverse of a field element must be described:

- Additive inverse: If $a \in \mathbb{F}_{2^m}$, then the additive inverse $(-a)$ of a in $\mathbb{F}_{2^m}$ is the unique solution to the equation $a + x = 0$ in $\mathbb{F}_{2^m}$. Note that $-a = a$ for all $a \in \mathbb{F}_{2^m}$.
- Multiplicative inverse: If $a \in \mathbb{F}_{2^m}$, $a \neq 0$, then the multiplicative inverse a^{-1} of a in $\mathbb{F}_{2^m}$ is the unique solution to the equation $ax = 1$ in $\mathbb{F}_{2^m}$.

Additive inverses and multiplicative inverses in $\mathbb{F}_{2^m}$ can be calculated efficiently. Multiplicative inverses can be calculated using the polynomial version of the extended Euclidean algorithm. Division and subtraction are defined in terms of additive and multiplicative inverses: $a - b$ in $\mathbb{F}_{2^m}$ is $a + (-b)$ in $\mathbb{F}_{2^m}$ and a/b in $\mathbb{F}_{2^m}$ is $a(b^{-1})$ in $\mathbb{F}_{2^m}$.

Here the characteristic 2 finite fields $\mathbb{F}_{2^m}$ used should have:

$$m \in \{163, 233, 239, 283, 409, 571\}$$

and addition and multiplication in $\mathbb{F}_{2^m}$ should be performed using one of the irreducible binary polynomials of degree m in Table 1. As before this restriction is designed to facilitate interoperability while enabling implementers to deploy efficient implementations capable of meeting common security requirements.

Field	Reduction Polynomial(s)
$\mathbb{F}_{2^{163}}$	$f(x) = x^{163} + x^7 + x^6 + x^3 + 1$
$\mathbb{F}_{2^{233}}$	$f(x) = x^{233} + x^{74} + 1$
$\mathbb{F}_{2^{239}}$	$f(x) = x^{239} + x^{36} + 1$ or $x^{239} + x^{158} + 1$
$\mathbb{F}_{2^{283}}$	$f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$
$\mathbb{F}_{2^{409}}$	$f(x) = x^{409} + x^{87} + 1$
$\mathbb{F}_{2^{571}}$	$f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$

Table 1: Representations of $\mathbb{F}_{2^m}$

The rule used to pick acceptable m's was: in each interval between integers in the set:

$$\{160, 224, 256, 384, 512, 1024\},$$

if such an m exists, select the smallest prime m in the interval with the property that there exists a Koblitz curve whose order is 2 or 4 times a prime over $\mathbb{F}_{2^m}$; otherwise simply select the smallest prime m in the interval. (A Koblitz curve is an elliptic curve over $\mathbb{F}_{2^m}$ with $a, b \in \{0, 1\}$, see Section 2.2.) The inclusion of $m = 239$ is an anomaly chosen since it has already been widely used in practice. The inclusion of $m = 283$ instead of $m = 277$ is an anomaly chosen to align this document with other standards efforts — in particular with the U.S. government's recommended elliptic curve domain parameters [186-2]. Composite m was avoided to align this specification with other standards efforts and to address concerns expressed by some experts about the security of elliptic curves defined over $\mathbb{F}_{2^m}$ with m composite — see, for example, [GS99, JMS01, GHS02, Hes05, MT06].

The rule used to pick acceptable reduction polynomials was: if a degree m binary irreducible trinomial:

$$f(x) = x^m + x^k + 1 \text{ with } m > k \geq 1$$

exists, use the irreducible trinomial with k as small as possible; otherwise use the degree m binary irreducible pentanomial:

$$f(x) = x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1 \text{ with } \ m > k_3 > k_2 > k_1 \geq 1$$

with (1) k_3 as small as possible, (2) k_2 as small as possible given k_3, and (3) k_1 as small as possible given k_3 and k_2. These polynomials enable efficient calculation of field operations. The second reduction polynomial with $m = 239$ is an anomaly chosen since it has been widely deployed.

2.2 Elliptic Curves

An elliptic curve over $\mathbb{F}_q$ is defined in terms of the solutions to an equation in $\mathbb{F}_q$. The form of the equation defining an elliptic curve over $\mathbb{F}_q$ differs depending on whether the field is a prime finite field or a characteristic 2 finite field.

Section 2.2.1 describes elliptic curves over prime finite fields, and Section 2.2.2 describes elliptic curves over characteristic 2 finite fields.

2.2.1 Elliptic Curves over $\mathbb{F}_p$

Let $\mathbb{F}_p$ be a prime finite field so that p is an odd prime number, and let $a, b \in \mathbb{F}_p$ satisfy $4a^3+27b^2 \not\equiv 0 \pmod{p}$. Then an elliptic curve $E(\mathbb{F}_p)$ over $\mathbb{F}_p$ defined by the parameters $a, b \in \mathbb{F}_p$ consists of the set of solutions or points $P = (x, y)$ for $x, y \in \mathbb{F}_p$ to the equation:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

together with an extra point $\mathcal{O}$ called the point at infinity. The equation $y^2 \equiv x^3+ax+b \pmod{p}$ is called the defining equation of $E(\mathbb{F}_p)$. For a given point $P = (x_P, y_P)$, x_P is called the x-coordinate of P, and y_P is called the y-coordinate of P.

The number of points on $E(\mathbb{F}_p)$ is denoted by $\#E(\mathbb{F}_p)$. The Hasse Theorem states that:

$$p + 1 - 2\sqrt{p} \leq \#E(\mathbb{F}_p) \leq p + 1 + 2\sqrt{p}.$$

It is possible to define an addition rule to add points on E. The addition rule is specified as follows:

1. Rule to add the point at infinity to itself:

$$\mathcal{O} + \mathcal{O} = \mathcal{O}.$$

2. Rule to add the point at infinity to any other point:

$$(x, y) + \mathcal{O} = \mathcal{O} + (x, y) = (x, y) \text{ for all } (x, y) \in E(\mathbb{F}_p).$$

3. Rule to add two points with the same x-coordinates when the points are either distinct or have y-coordinate 0:

$$(x, y) + (x, -y) = \mathcal{O} \text{ for all } (x, y) \in E(\mathbb{F}_p)$$

— i.e. the negative of the point (x, y) is $-(x, y) = (x, -y)$.

4. Rule to add two points with different x-coordinates: Let $(x_1, y_1) \in E(\mathbb{F}_p)$ and $(x_2, y_2) \in E(\mathbb{F}_p)$ be two points such that $x_1 \neq x_2$. Then $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$, where:

$$x_3 \equiv \lambda^2 - x_1 - x_2 \pmod{p},\ y_3 \equiv \lambda(x_1 - x_3) - y_1 \pmod{p}, \text{ and } \lambda \equiv \frac{y_2 - y_1}{x_2 - x_1} \pmod{p}.$$

5. Rule to add a point to itself (double a point): Let $(x_1, y_1) \in E(\mathbb{F}_p)$ be a point with $y_1 \neq 0$. Then $(x_1, y_1) + (x_1, y_1) = (x_3, y_3)$, where:

$$x_3 \equiv \lambda^2 - 2x_1 \pmod{p},\ y_3 \equiv \lambda(x_1 - x_3) - y_1 \pmod{p}, \text{ and } \lambda \equiv \frac{3x_1^2 + a}{2y_1} \pmod{p}.$$

The set of points on $E(\mathbb{F}_p)$ forms a group under this addition rule. Furthermore the group is abelian — meaning that $P_1 + P_2 = P_2 + P_1$ for all points $P_1, P_2 \in E(\mathbb{F}_p)$. Notice that the addition rule can always be computed efficiently using simple field arithmetic.

Cryptographic schemes based on ECC rely on scalar multiplication of elliptic curve points. Given an integer k and a point $P \in E(\mathbb{F}_p)$, scalar multiplication is the process of adding P to itself k times. The result of this scalar multiplication is denoted kP. Scalar multiplication of elliptic curve points can be computed efficiently using the addition rule together with the double-and-add algorithm or one of its variants.

2.2.2 Elliptic Curves over $\mathbb{F}_{2^m}$

Let $\mathbb{F}_{2^m}$ be a characteristic 2 finite field, and let $a, b \in \mathbb{F}_{2^m}$ satisfy $b \neq 0$ in $\mathbb{F}_{2^m}$. Then a (non-supersingular) elliptic curve $E(\mathbb{F}_{2^m})$ over $\mathbb{F}_{2^m}$ defined by the parameters $a, b \in \mathbb{F}_{2^m}$ consists of the set of solutions or points $P = (x, y)$ for $x, y \in \mathbb{F}_{2^m}$ to the equation:

$$y^2 + xy = x^3 + ax^2 + b \text{ in } \mathbb{F}_{2^m}$$

together with an extra point $\mathcal{O}$ called the point at infinity. (Here the only elliptic curves over $\mathbb{F}_{2^m}$ of interest are non-supersingular elliptic curves.)

The number of points on $E(\mathbb{F}_{2^m})$ is denoted by $\#E(\mathbb{F}_{2^m})$. The Hasse Theorem states that:

$$2^m + 1 - 2\sqrt{2^m} \leq \#E(\mathbb{F}_{2^m}) \leq 2^m + 1 + 2\sqrt{2^m}.$$

It is again possible to define an addition rule to add points on E as it was in Section 2.2.1. The addition rule is specified as follows:

1. Rule to add the point at infinity to itself:

$$\mathcal{O} + \mathcal{O} = \mathcal{O}.$$

2. Rule to add the point at infinity to any other point:

$$(x, y) + \mathcal{O} = \mathcal{O} + (x, y) = (x, y) \text{ for all } (x, y) \in E(\mathbb{F}_p).$$

3. Rule to add two points with the same x-coordinates when the points are either distinct or have x-coordinate 0:

$$(x, y) + (x, x + y) = \mathcal{O} \text{ for all } (x, y) \in E(\mathbb{F}_p)$$

   — i.e. the negative of the point (x, y) is $-(x, y) = (x, x + y)$.

4. Rule to add two points with different x-coordinates: Let $(x_1, y_1) \in E(\mathbb{F}_{2^m})$ and $(x_2, y_2) \in E(\mathbb{F}_{2^m})$ be two points such that $x_1 \neq x_2$. Then $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$, where:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \text{ in } \mathbb{F}_{2^m},\ y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \text{ in } \mathbb{F}_{2^m}, \text{ and } \lambda = \frac{y_1 + y_2}{x_1 + x_2} \text{ in } \mathbb{F}_{2^m}.$$

5. Rule to add a point to itself (double a point): Let $(x_1, y_1) \in E(\mathbb{F}_{2^m})$ be a point with $x_1 \neq 0$. Then $(x_1, y_1) + (x_1, y_1) = (x_3, y_3)$, where:

$$x_3 = \lambda^2 + \lambda + a \text{ in } \mathbb{F}_{2^m},\ y_3 = x_1^2 + (\lambda + 1)x_3 \text{ in } \mathbb{F}_{2^m}, \text{ and } \lambda = x_1 + \frac{y_1}{x_1} \text{ in } \mathbb{F}_{2^m}.$$

The set of points on $E(\mathbb{F}_{2^m})$ forms an abelian group under this addition rule. Notice that the addition rule can always be computed efficiently using simple field arithmetic.

Cryptographic schemes based on ECC rely on scalar multiplication of elliptic curve points. As before given an integer k and a point $P \in E(\mathbb{F}_{2^m})$, scalar multiplication is the process of adding P to itself k times. The result of this scalar multiplication is denoted kP.

2.3 Data Types and Conversions

The schemes specified in this document involve operations using several different data types. This section lists the different data types and describes how to convert one data type to another.

Five data types are employed in this document: three types associated with elliptic curve arithmetic — integers, field elements, and elliptic curve points — as well as octet strings which are used to communicate and store information, and bit strings which are used by some of the primitives.

Frequently it is necessary to convert one of the data types into another — for example to represent an elliptic curve point as an octet string. The remainder of this section is devoted to describing how the necessary conversions should be performed.

Figure 1 illustrates which conversions are needed and where they are described.

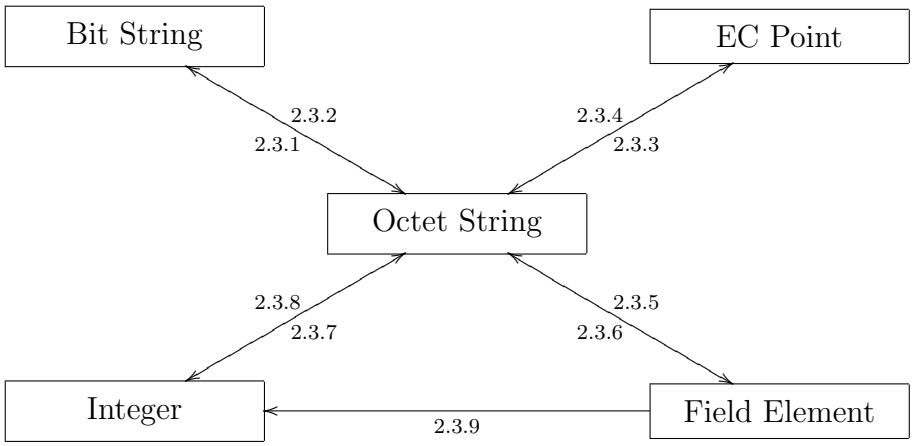

Figure 1: Converting between Data Types

2.3.1 Bit-String-to-Octet-String Conversion

Bit strings should be converted to octet strings as described in this section. Informally the idea is to pad the bit string with 0's on the left to make its length a multiple of 8, then chop the result up into octets. Formally the conversion routine is specified as follows:

Input: A bit string B of length $blen$ bits.

Output: An octet string M of length $mlen = \lceil blen/8 \rceil$ octets.

Actions: Convert the bit string $B = B_0 B_1 \ldots B_{blen-1}$ to an octet string $M = M_0 M_1 \ldots M_{mlen-1}$ as follows:

1. For $0 < i \leq mlen - 1$, let:

$$M_i = B_{blen-8-8(mlen-1-i)} B_{blen-7-8(mlen-1-i)} \ldots B_{blen-1-8(mlen-1-i)}.$$

2. Let M_0 have its leftmost $8(mlen) - blen$ bits set to 0, and its rightmost $8 - (8(mlen) - blen)$ bits set to $B_0 B_1 \ldots B_{8-8(mlen)+blen-1}$.

3. Output M.

2.3.2 Octet-String-to-Bit-String Conversion

Octet strings should be converted to bit strings as described in this section. Informally the idea is simply to view the octet string as a bit string instead. Formally the conversion routine is specified as follows:

Input: An octet string M of length $mlen$ octets.

Output: A bit string B of length $blen = 8(mlen)$ bits.

Actions: Convert the octet string $M = M_0 M_1 \dots M_{mlen-1}$ to a bit string $B = B_0 B_1 \dots B_{blen-1}$ as follows:

1. For $0 \leq i \leq mlen - 1$, set:
$$B_{8i} B_{8i+1} \dots B_{8i+7} = M_i.$$
2. Output B.

2.3.3 Elliptic-Curve-Point-to-Octet-String Conversion

Elliptic curve points should be converted to octet strings as described in this section. Informally, if point compression is being used, the idea is that the compressed y-coordinate is placed in the leftmost octet of the octet string along with an indication that point compression is on, and the x-coordinate is placed in the remainder of the octet string; otherwise if point compression is off, the leftmost octet indicates that point compression is off, and the remainder of the octet string contains the x-coordinate followed by the y-coordinate. Formally the conversion routine is specified as follows:

Setup: Decide whether or not to represent points using point compression.

Input: A point P on an elliptic curve over $\mathbb{F}_q$ defined by the field elements a, b.

Output: An octet string M of length $mlen$ octets where $mlen = 1$ if $P = \mathcal{O}$, $mlen = \lceil (\log_2 q)/8 \rceil + 1$ if $P \neq \mathcal{O}$ and point compression is used, and $mlen = 2\lceil (\log_2 q)/8 \rceil + 1$ if $P \neq \mathcal{O}$ and point compression is not used.

Actions: Convert P to an octet string $M = M_0 M_1 \dots M_{mlen-1}$ as follows:

1. If $P = \mathcal{O}$, output $M = 00_{16}$.
2. If $P = (x_P, y_P) \neq \mathcal{O}$ and point compression is being used, proceed as follows:
   2.1. Convert the field element x_P to an octet string X of length $\lceil (\log_2 q)/8 \rceil$ octets using the conversion routine specified in Section 2.3.5.
   2.2. Derive from y_P a single bit $\tilde{y}_P$ as follows (this allows the y-coordinate to be represented compactly using a single bit):
      2.2.1. If $q = p$ is an odd prime, set $\tilde{y}_P = y_P \pmod 2$.
      2.2.2. If $q = 2^m$, set $\tilde{y}_P = 0$ if $x_P = 0$, otherwise compute $z = z_{m-1} x^{m-1} + \dots + z_1 x + z_0$ such that $z = y_P x_P^{-1}$ and set $\tilde{y}_P = z_0$.

2.3. Assign the value 02_{16} to the single octet Y if $\tilde{y}_P = 0$, or the value 03_{16} if $\tilde{y}_P = 1$.

2.4. Output $M = Y \,\|\, X$.

3. If $P = (x_P, y_P) \neq \mathcal{O}$ and point compression is not being used, proceed as follows:

3.1. Convert the field element x_P to an octet string X of length $\lceil (\log_2 q)/8 \rceil$ octets using the conversion routine specified in Section 2.3.5.

3.2. Convert the field element y_P to an octet string Y of length $\lceil (\log_2 q)/8 \rceil$ octets using the conversion routine specified in Section 2.3.5.

3.3. Output $M = 04_{16} \,\|\, X \,\|\, Y$.

2.3.4 Octet-String-to-Elliptic-Curve-Point Conversion

Octet strings should be converted to elliptic curve points as described in this section. Informally the idea is that, if the octet string represents a compressed point, the compressed y-coordinate is recovered from the leftmost octet, the x-coordinate is recovered from the remainder of the octet string, and then the point compression process is reversed; otherwise the leftmost octet of the octet string is removed, the x-coordinate is recovered from the left half of the remaining octet string, and the y-coordinate is recovered from the right half of the remaining octet string. Formally the conversion routine is specified as follows:

Input: An elliptic curve over $\mathbb{F}_q$ defined by the field elements a, b, and an octet string M which is either the single octet 00_{16}, an octet string of length $mlen = \lceil (\log_2 q)/8 \rceil + 1$, or an octet string of length $mlen = 2\lceil (\log_2 q)/8 \rceil + 1$.

Output: An elliptic curve point P, or "invalid".

Actions: Convert M to an elliptic curve point P as follows:

1. If $M = 00_{16}$, output $P = \mathcal{O}$.

2. If M has length $\lceil (\log_2 q)/8 \rceil + 1$ octets, proceed as follows:

2.1. Parse $M = Y \,\|\, X$ as a single octet Y followed by $\lceil (\log_2 q)/8 \rceil$ octets X.

2.2. Convert X to a field element x_P of $\mathbb{F}_q$ using the conversion routine specified in Section 2.3.6. Output "invalid" and stop if the routine outputs "invalid".

2.3. If $Y = 02_{16}$, set $\tilde{y}_P = 0$, and if $Y = 03_{16}$, set $\tilde{y}_P = 1$. Otherwise output "invalid" and stop.

2.4. Derive from x_P and $\tilde{y}_P$ an elliptic curve point $P = (x_P, y_P)$, where:

2.4.1. If $q = p$ is an odd prime, compute the field element $\alpha \equiv x_P^3 + a x_P + b \pmod{p}$, and compute a square root β of α modulo p. Output "invalid" and stop if there are no square roots of α modulo p, otherwise set $y_P = \beta$ if $\beta \equiv \tilde{y}_P \pmod{2}$, and set $y_P = p - \beta$ if $\beta \not\equiv \tilde{y}_P \pmod{2}$.

2.4.2. If $q = 2^m$ and $x_P = 0$, output $y_P = b^{2^{m-1}}$ in $\mathbb{F}_{2^m}$.

2.4.3. If $q = 2^m$ and $x_P \neq 0$, compute the field element $\beta = x_P + a + bx_P^{-2}$ in $\mathbb{F}_{2^m}$, and find an element $z = z_{m-1}x^{m-1} + \cdots + z_1 x + z_0$ such that $z^2 + z = \beta$ in $\mathbb{F}_{2^m}$. Output "invalid" and stop if no such z exists, otherwise set $y_P = x_P z$ in $\mathbb{F}_{2^m}$ if $z_0 = \tilde{y}_P$, and set $y_P = x_P(z+1)$ in $\mathbb{F}_{2^m}$ if $z_0 \neq \tilde{y}_P$.

2.5. Output $P = (x_P, y_P)$.

3. If M has length $2\lceil(\log_2 q)/8\rceil + 1$ octets, proceed as follows:

3.1. Parse $M = W \,\|\, X \,\|\, Y$ as a single octet W followed by $\lceil(\log_2 q)/8\rceil$ octets X followed by $\lceil(\log_2 q)/8\rceil$ octets Y.

3.2. Check that $W = 04_{16}$. If $W \neq 04_{16}$, output "invalid" and stop.

3.3. Convert X to a field element x_P of $\mathbb{F}_q$ using the conversion routine specified in Section 2.3.6. Output "invalid" and stop if the routine outputs "invalid".

3.4. Convert Y to a field element y_P of $\mathbb{F}_q$ using the conversion routine specified in Section 2.3.6. Output "invalid" and stop if the routine outputs "invalid".

3.5. Check that $P = (x_P, y_P)$ satisfies the defining equation of the elliptic curve.

3.6. Output $P = (x_P, y_P)$.

2.3.5 Field-Element-to-Octet-String Conversion

Field elements should be converted to octet strings as described in this section. Informally the idea is that, if the field is $\mathbb{F}_p$, convert the integer to an octet string, and if the field is $\mathbb{F}_{2^m}$, view the coefficients of the polynomial as a bit string with the highest degree term on the left and convert the bit string to an octet string. Formally the conversion routine is specified as follows:

Input: An element a of the field $\mathbb{F}_q$.

Output: An octet string M of length $mlen = \lceil(\log_2 q)/8\rceil$ octets.

Actions: Convert a to an octet string $M = M_0 M_1 \ldots M_{mlen-1}$ as follows:

1. If $q = p$ is an odd prime, then a is an integer in the interval $[0, p-1]$. Convert a to M using the conversion routine specified in Section 2.3.7 (with a and $mlen$ as inputs). Output M.

2. If $q = 2^m$, then $a = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0$ is a binary polynomial. Convert a to M as follows:

2.1. For $0 < i \leq mlen - 1$, let:

$$M_i = a_{7+8(mlen-1-i)} a_{6+8(mlen-1-i)} \ldots a_{8(mlen-1-i)}.$$

2.2. Let M_0 have its leftmost $8(mlen) - m$ bits set to 0, and its rightmost $8 - (8(mlen) - m)$ bits set to $a_{m-1}a_{m-2}\ldots a_{8(mlen)-8}$.

2.3. Output M.

2.3.6 Octet-String-to-Field-Element Conversion

Octet strings should be converted to field elements as described in this section. Informally the idea is that, if the field is $\mathbb{F}_p$, convert the octet string to an integer, and if the field is $\mathbb{F}_{2^m}$, use the bits of the octet string as the coefficients of the binary polynomial with the rightmost bit as the constant term. Formally the conversion routine is specified as follows:

Input: An indication of the field $\mathbb{F}_q$ used and an octet string M of length $mlen = \lceil (\log_2 q)/8 \rceil$ octets.

Output: An element a in $\mathbb{F}_q$, or "invalid".

Actions: Convert $M = M_0 M_1 \ldots M_{mlen-1}$ with $M_i = M_i^0 M_i^1 \ldots M_i^7$ to a field element a as follows:

1. If $q = p$ is an odd prime, then a needs to be an integer in the interval $[0, p-1]$. Convert M to an integer a using the conversion routine specified in Section 2.3.8. Output "invalid" and stop if a does not lie in the interval $[0, p-1]$, otherwise output a.

2. If $q = 2^m$, then a needs to be a binary polynomial of degree $m-1$ or less. Set the field element a to be $a = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0$ with:

$$a_i = M_{mlen-1-\lfloor i/8 \rfloor}^{7-i+8(\lfloor i/8 \rfloor)}.$$

   Output "invalid" and stop if the leftmost $8(mlen) - m$ bits of M_0 are not all 0, otherwise output a.

2.3.7 Integer-to-Octet-String Conversion

Integers should be converted to octet strings as described in this section. Informally the idea is to represent the integer in binary then convert the resulting bit string to an octet string. Formally the conversion routine is specified as follows:

Input: A non-negative integer x together with the desired length $mlen$ of the octet string. It must be the case that:

$$2^{8(mlen)} > x.$$

Output: An octet string M of length $mlen$ octets.

Actions: Convert $x = x_{mlen-1}2^{8(mlen-1)} + x_{mlen-2}2^{8(mlen-2)} + \cdots + x_1 2^8 + x_0$ represented in base $2^8 = 256$ to an octet string $M = M_0 M_1 \ldots M_{mlen-1}$ as follows:

1. For $0 \le i \le mlen - 1$, set:

$$M_i = x_{mlen-1-i}.$$

2. Output M.

2.3.8 Octet-String-to-Integer Conversion

Octet strings should be converted to integers as described in this section. Informally the idea is simply to view the octet string as the base 256 representation of the integer. Formally the conversion routine is specified as follows:

Input: An octet string M of length *mlen* octets.

Output: An integer x.

Actions: Convert $M = M_0 M_1 \dots M_{mlen-1}$ to an integer x as follows:

1. View M_i as an integer in the range $[0, 255]$ and set:
$$x = \sum_{i=0}^{mlen-1} 2^{8(mlen-1-i)} M_i.$$
2. Output x.

2.3.9 Field-Element-to-Integer Conversion

Field elements should be converted to integers as described in this section. Informally the idea is that, if the field is $\mathbb{F}_p$ no conversion is required, and if the field is $\mathbb{F}_{2^m}$ first convert the binary polynomial to a bit string then convert the bit string to an integer. Formally the conversion routine is specified as follows:

Input: An element a of the field $\mathbb{F}_q$.

Output: An integer x.

Actions: Convert the field element a to an integer x as follows:

1. If $q = p$ is an odd prime, then a must be an integer in the interval $[0, p-1]$. Output $x = a$.
2. If $q = 2^m$, then a must be a binary polynomial of degree at most $m-1$ — i.e. $a = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0$. Set:
$$x = \sum_{i=0}^{m-1} 2^i a_i.$$
Output x.

3 Cryptographic Components

This section describes the various cryptographic components that are used to build signature schemes, encryption schemes, and key agreement schemes later in this document.

See Appendix B for a commentary on the contents on this section, including implementation discussion, security discussion, and references.

3.1 Elliptic Curve Domain Parameters

The operation of each of the public-key cryptographic schemes described in this document involves arithmetic operations on an elliptic curve over a finite field determined by some elliptic curve domain parameters.

This section addresses the provision of elliptic curve domain parameters. It describes what elliptic curve domain parameters are, how they should be generated, and how they should be validated.

Two types of elliptic curve domain parameters may be used: elliptic curve domain parameters over $\mathbb{F}_p$, and elliptic curve domain parameters over $\mathbb{F}_{2^m}$. Section 3.1.1 describes elliptic curve domain parameters over $\mathbb{F}_p$, and Section 3.1.2 describes elliptic curve domain parameters over $\mathbb{F}_{2^m}$.

Elliptic curve domain parameters can be verifiably random, which means that the parameters are obtained in part as the output of a secure hash function, applied to some seed value S. Verifiably random elliptic domain parameters are recommended, but others may be used for various reasons, such as superior performance.

3.1.1 Elliptic Curve Domain Parameters over $\mathbb{F}_p$

Elliptic curve domain parameters over $\mathbb{F}_p$ are a sextuple:

$$T = (p, a, b, G, n, h)$$

consisting of an integer p specifying the finite field $\mathbb{F}_p$, two elements $a, b \in \mathbb{F}_p$ specifying an elliptic curve $E(\mathbb{F}_p)$ defined by the equation:

$$E: \ y^2 \equiv x^3 + ax + b \pmod{p},$$

a base point $G = (x_G, y_G)$ on $E(\mathbb{F}_p)$, a prime n which is the order of G, and an integer h which is the cofactor $h = \#E(\mathbb{F}_p)/n$.

Elliptic curve domain parameters over $\mathbb{F}_p$ precisely specify an elliptic curve and base point. This is necessary to precisely define public-key cryptographic schemes based on ECC.

If the elliptic curve domain parameters T are verifiably random, as specified in Section 3.1.3, then they should be accompanied by the seed value S from which they are derived.

Section 3.1.1.1 describes how to generate elliptic curve domain parameters over $\mathbb{F}_p$, and Section 3.1.1.2 describes how to validate elliptic curve domain parameters over $\mathbb{F}_p$.

3.1.1.1 Elliptic Curve Domain Parameters over $\mathbb{F}_p$ Generation Primitive

Elliptic curve domain parameters over $\mathbb{F}_p$ should be generated as follows:

Input: The approximate security level in bits required from the elliptic curve domain parameters — this must be an integer $t \in \{80, 112, 128, 192, 256\}$. Optionally, a seed value S.

Output: Elliptic curve domain parameters over $\mathbb{F}_p$:

$$T = (p, a, b, G, n, h)$$

such that taking logarithms on the associated elliptic curve requires approximately 2^t operations.

Actions: Generate elliptic curve domain parameters over $\mathbb{F}_p$ as follows:

1. Select a prime p such that $\lceil \log_2 p \rceil = 2t$ if $80 < t < 256$, such that $\lceil \log_2 p \rceil = 521$ if $t = 256$, and such that $\lceil \log_2 p \rceil = 192$ if $t = 80$ to determine the finite field $\mathbb{F}_p$.

2. Select elements $a, b \in \mathbb{F}_p$ to determine the elliptic curve $E(\mathbb{F}_p)$ defined by the equation:

   $$E: \ y^2 \equiv x^3 + ax + b \pmod{p},$$

   a base point $G = (x_G, y_G)$ on $E(\mathbb{F}_p)$, a prime n which is the order of G, and an integer h which is the cofactor $h = \#E(\mathbb{F}_p)/n$, subject to the following constraints:

   - $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$.
   - $\#E(\mathbb{F}_p) \neq p$.
   - $p^B \not\equiv 1 \pmod{n}$ for all $1 \leq B < 100$.
   - $h \leq 2^{t/8}$.
   - $n - 1$ and $n + 1$ should each have a large prime factor r, which is large in the sense that $\log_n(r) > \frac{19}{20}$.

   If seed S is provided, then the coefficient pair (a, b), or the point G, or both, should be derived from S. See Section 3.1.3.

3. Output $T = (p, a, b, G, n, h)$.

This primitive allows any of the known curve selection methods to be used — for example the methods based on complex multiplication and the methods based on general point counting algorithms. However to foster interoperability it is strongly recommended that implementers use one of the elliptic curve domain parameters over $\mathbb{F}_p$ specified in SEC 2 [SEC 2]. See Appendix B for further discussion.

3.1.1.2 Validation of Elliptic Curve Domain Parameters over $\mathbb{F}_p$

Frequently, it is either necessary or desirable for an entity using elliptic curve domain parameters over $\mathbb{F}_p$ to receive an assurance that the parameters are valid — that is, that they satisfy the arithmetic requirements of elliptic curve domain parameters — either to prevent malicious insertion of insecure parameters, or to detect inadvertent coding or transmission errors.

There are four acceptable methods for an entity U to receive an assurance that elliptic curve domain parameters over $\mathbb{F}_p$ are valid. At least one of the methods must be supplied, although in many cases greater security may be obtained by carrying out more than one of the methods.

The four acceptable methods are:

1. Entity U performs validation of the elliptic curve domain parameters over $\mathbb{F}_p$ itself using the validation primitive described in Section 3.1.1.2.1.
2. Entity U generates the elliptic curve domain parameters over $\mathbb{F}_p$ itself using a trusted system that in turn uses the primitive specified in Section 3.1.1.1.
3. Entity U receives assurance in an authentic manner that a party trusted with respect to entity U's use of the elliptic curve domain parameters over $\mathbb{F}_p$ has performed validation of the parameters using the validation primitive described in Section 3.1.1.2.1.
4. Entity U receives assurance in an authentic manner that a party trusted with respect to entity U's use of the elliptic curve domain parameters over $\mathbb{F}_p$ generated the parameters using a trusted system that in turn uses the primitive specified in Section 3.1.1.1.

Usually when entity U accepts another party's assurance that elliptic curve domain parameters are valid, the other party is a CA.

3.1.1.2.1 Elliptic Curve Domain Parameters over $\mathbb{F}_p$ Validation Primitive

The elliptic curve domain parameters over $\mathbb{F}_p$ validation primitive should be used to check that elliptic curve domain parameters over $\mathbb{F}_p$ are valid as follows:

Input: Elliptic curve domain parameters over $\mathbb{F}_p$:

$$T = (p, a, b, G, n, h),$$

along with an integer $t \in \{80, 112, 128, 192, 256\}$ which is the approximate security level in bits required from the elliptic curve domain parameters. Optionally, a seed value S.

Output: An indication of whether the elliptic curve domain parameters are valid or not — either "valid" or "invalid".

Actions: Validate the elliptic curve domain parameters over $\mathbb{F}_p$ as follows:

1. Check that p is an odd prime such that $\lceil \log_2 p \rceil = 2t$ if $80 < t < 256$, or such that $\lceil \log_2 p \rceil = 521$ if $t = 256$, or such that $\lceil \log_2 p \rceil = 192$ if $t = 80$.
2. Check that a, b, x_G, and y_G are integers in the interval $[0, p-1]$.
3. Check that $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$.
4. Check that $y_G^2 \equiv x_G^3 + ax_G + b \pmod{p}$.
5. Check that n is prime.

6. Check that $h \leq 2^{t/8}$, and that $h = \lfloor(\sqrt{p}+1)^2/n\rfloor$.
7. Check that $nG = \mathcal{O}$.
8. Check that $p^B \not\equiv 1 \pmod{n}$ for all $1 \leq B < 100$, and that $n \neq p$.
9. If any of the checks fail, output "invalid", otherwise output "valid".

Step 8 above excludes the known weak classes of curves which are susceptible to either the Menezes-Okamoto-Vanstone attack, or the Frey-Rück attack, or the Semaev-Smart-Satoh-Araki attack. See Appendix B for further discussion.

If the elliptic curve domain parameters have been generated verifiably at random as described in Section 3.1.3, it may also be checked that a and b have been correctly derived from the seed S or that G has been correctly derived from the seed S, or all have.

3.1.2 Elliptic Curve Domain Parameters over $\mathbb{F}_{2^m}$

Elliptic curve domain parameters over $\mathbb{F}_{2^m}$ are a septuple:

$$T = (m, f(x), a, b, G, n, h)$$

consisting of an integer m specifying the finite field $\mathbb{F}_{2^m}$, an irreducible binary polynomial $f(x)$ of degree m specifying the representation of $\mathbb{F}_{2^m}$, two elements $a, b \in \mathbb{F}_{2^m}$ specifying the elliptic curve $E(\mathbb{F}_{2^m})$ defined by the equation:

$$y^2 + xy = x^3 + ax^2 + b \text{ in } \mathbb{F}_{2^m},$$

a base point $G = (x_G, y_G)$ on $E(\mathbb{F}_{2^m})$, a prime n which is the order of G, and an integer h which is the cofactor $h = \#E(\mathbb{F}_{2^m})/n$.

Elliptic curve domain parameters over $\mathbb{F}_{2^m}$ precisely specify an elliptic curve and base point. This is necessary to precisely define public-key cryptographic schemes based on ECC.

If the elliptic curve domain parameters T are verifiably random, as specified in Section 3.1.3, then they should be accompanied by the seed value S from which they are derived.

Section 3.1.2.1 describes how to generate elliptic curve domain parameters over $\mathbb{F}_{2^m}$, and Section 3.1.2.2 describes how to validate elliptic curve domain parameters over $\mathbb{F}_{2^m}$.

3.1.2.1 Elliptic Curve Domain Parameters over $\mathbb{F}_{2^m}$ Generation Primitive

Elliptic curve domain parameters over $\mathbb{F}_{2^m}$ should be generated as follows:

Input: The approximate security level in bits required from the elliptic curve domain parameters — this must be an integer $t \in \{80, 112, 128, 192, 256\}$. Optionally, a seed value S.

Output: Elliptic curve domain parameters over $\mathbb{F}_{2^m}$:

$$T = (m, f(x), a, b, G, n, h)$$

such that taking logarithms on the associated elliptic curve requires approximately 2^t operations.

Actions: Generate elliptic curve domain parameters over $\mathbb{F}_{2^m}$ as follows:

1. Let t' denote the smallest integer greater than t in the set $\{112, 128, 192, 256, 512\}$. Select $m \in \{163, 233, 239, 283, 409, 571\}$ such that $2t < m < 2t'$ to determine the finite field $\mathbb{F}_{2^m}$.

2. Select a binary irreducible polynomial $f(x)$ of degree m from Table 1 in Section 2.1.2 to determine the representation of $\mathbb{F}_{2^m}$.

3. Select elements $a, b \in \mathbb{F}_{2^m}$ to determine the elliptic curve $E(\mathbb{F}_{2^m})$ defined by the equation:

$$E: \ y^2 + xy = x^3 + ax^2 + b \text{ in } \mathbb{F}_{2^m},$$

   a base point $G = (x_G, y_G)$ on $E(\mathbb{F}_{2^m})$, a prime n which is the order of G, and an integer h which is the cofactor $h = \#E(\mathbb{F}_{2^m})/n$, subject to the following constraints:

   - $b \neq 0$ in $\mathbb{F}_{2^m}$.
   - $\#E(\mathbb{F}_{2^m}) \neq 2^m$.
   - $2^B \not\equiv 1 \pmod{n}$ for all $1 \leq B < 100m$.
   - $h \leq 2^{t/8}$.
   - $n-1$ and $n+1$ should each have a large prime factor r, which is large in the sense that $\log_n(r) > \frac{19}{20}$.

   If seed value S is provided, then coefficient pair (a, b), or point G, or both, should be derived from it. See Section 3.1.3.

4. Output $T = (m, f(x), a, b, G, n, h)$.

This primitive also allows any of the known curve selection methods to be used — for example the methods based on complex multiplication and the methods based on general point counting algorithms. However to foster interoperability it is strongly recommended that implementers use one of the recommended elliptic curve domain parameters over $\mathbb{F}_{2^m}$ specified in SEC 2 [SEC 2]. See Appendix B for further discussion.

3.1.2.2 Validation of Elliptic Curve Domain Parameters over $\mathbb{F}_{2^m}$

Frequently, it is either necessary or desirable for an entity using elliptic curve domain parameters over $\mathbb{F}_{2^m}$ to receive an assurance that the parameters are valid — that is, that they satisfy the arithmetic requirements of elliptic curve domain parameters — either to prevent malicious insertion of insecure parameters, or to detect inadvertent coding or transmission errors.

There are four acceptable methods for an entity U to receive an assurance that elliptic curve domain parameters over $\mathbb{F}_{2^m}$ are valid. At least one of the methods must be supplied, although in many cases greater security may be obtained by carrying out more than one of the methods.

The four acceptable methods are:

1. Entity U performs validation of the elliptic curve domain parameters over $\mathbb{F}_{2^m}$ itself using the validation primitive described in Section 3.1.2.2.1.

2. Entity U generates the elliptic curve domain parameters over $\mathbb{F}_{2^m}$ itself using a trusted system that in turn uses the primitive specified in Section 3.1.2.1.

3. Entity U receives assurance in an authentic manner that a party trusted with respect to entity U's use of the elliptic curve domain parameters over $\mathbb{F}_{2^m}$ has performed validation of the parameters using the validation primitive described in Section 3.1.2.2.1.

4. Entity U receives assurance in an authentic manner that a party trusted with respect to entity U's use of the elliptic curve domain parameters over $\mathbb{F}_{2^m}$ generated the parameters using a trusted system that in turn uses the primitive specified in Section 3.1.2.1.

Usually when entity U accepts another party's assurance that elliptic curve domain parameters are valid, the other party is a CA.

3.1.2.2.1 Elliptic Curve Domain Parameters over $\mathbb{F}_{2^m}$ Validation Primitive

The elliptic curve domain parameters over $\mathbb{F}_{2^m}$ validation primitive should be used to check that elliptic curve domain parameters over $\mathbb{F}_{2^m}$ are valid as follows:

Input: Elliptic curve domain parameters over $\mathbb{F}_{2^m}$:

$$T = (m, f(x), a, b, G, n, h)$$

along with an integer $t \in \{80, 112, 128, 192, 256\}$ which is the approximate security level in bits required from the elliptic curve domain parameters.

Output: An indication of whether the elliptic curve domain parameters are valid or not — either "valid" or "invalid".

Actions: Validate the elliptic curve domain parameters over $\mathbb{F}_{2^m}$ as follows:

1. Let t' denote the smallest integer greater than t in the set $\{112, 128, 192, 256, 512\}$. Check that m is an integer in the set $\{163, 233, 239, 283, 409, 571\}$ such that $2t < m < 2t'$.

2. Check that $f(x)$ is a binary irreducible polynomial of degree m which is listed in Table 1 in Section 2.1.2.

3. Check that a, b, x_G, and y_G are binary polynomials of degree $m - 1$ or less.

4. Check that $b \neq 0$ in $\mathbb{F}_{2^m}$.

5. Check that $y_G^2 + x_G y_G = x_G^3 + a x_G^2 + b$ in $\mathbb{F}_{2^m}$.

6. Check that n is prime.

7. Check that $h \leq 2^{t/8}$, and that $h = \lfloor (\sqrt{2^m} + 1)^2 / n \rfloor$.

8. Check that $nG = \mathcal{O}$.

9. Check that $2^B \not\equiv 1 \pmod{n}$ for all $1 \leq B < 100m$, and that $nh \neq 2^m$.

10. If any of the checks fail, output "invalid", otherwise output "valid".

Step 9 above excludes the known weak classes of curves which are susceptible to either the Menezes-Okamoto-Vanstone attack, or the Frey-Rück attack, or the Semaev-Smart-Satoh-Araki attack. See Appendix B for further discussion.

If the elliptic curve domain parameters have been generated verifiably at random as described in Section 3.1.3, it may also be checked that a and b have been correctly derived from the seed, and it may also checked that G has been correctly derived from S.

3.1.3 Verifiably Random Curves and Base Point Generators

The section specifies how to derive from a seed S the elliptic curve coefficients a and b, and the base point generator G. These methods are consistent with ANS X9.62 [X9.62b].

The two routines here can be used for both (a) generating a verifiably random elliptic curve or base point, and (b) verifying that an elliptic curve or a base point is verifiably random. In the first application, the user selects the seed and performs the selection routine. In the second routine, the user is given the seed from another user who generated the elliptic curve or base point. The user then re-runs the routine either to recover the elliptic curve or base point, or to check if the result equals the existing elliptic curve or base point which is the one intended for use.

3.1.3.1 Curve Selection

Input: A "seed" octet string S of length $g/8$ octets, field size q, hash function *Hash* of output length *hashlen* octets, and field element $a \in \mathbb{F}_q$.

Output: A field element $b \in \mathbb{F}_q$ or "failure".

Actions: Generate the element b as follows:

1. Let $m = \lceil \log_2 q \rceil$.
2. Let $t = 8hashlen$.
3. Let $s = \lfloor (m-1)/t \rfloor$.
4. Let $k = m - st$ if q is even, and let $k = m - st - 1$ if q is odd.
5. Convert S to an integer s_0.
6. For j from 0 to s, do the following:
   6.1. Let $s_j = s_0 + j \bmod 2^g$.
   6.2. Let S_j be the integer s_j converted to a octet string of length $g/8$ octets.
   6.3. Let $H_j = Hash(S_j)$.
   6.4. Convert H_j to an integer e_j.

7. Let $e = e_0 2^{ts} + e_1 2^{t(s-1)} + \cdots + e_s \bmod 2^{k+st}$.
8. Convert e to a field element as follows:
   8.1. Convert e to an octet string E of length $mlen = \lceil (\log_2 q)/8 \rceil$ octets.
   8.2. Convert E to a field element $r \in \mathbb{F}_q$.
9. If q is even, then do as follows:
   9.1. If $r = 0$, then output "failure" and stop.
   9.2. If $r \neq 0$, then output $b = r \in \mathbb{F}_q$ and stop.
10. If q is odd, then do as follows:
    10.1. If $a = 0$, then output "failure" and stop.
    10.2. If $4r + 27 = 0$ in $\mathbb{F}_q$, then output "failure" and stop.
    10.3. If a^3/r does not have a square root in $\mathbb{F}_q$, then output "failure" and stop.
    10.4. Otherwise, output $b = \sqrt{a^3/r} \in \mathbb{F}_q$ and stop.

3.1.3.2 Point Selection

Input: A "seed" octet string S of length $g/8$ octets, field size q, hash function *Hash* of output length *hashlen*, and elliptic curve parameters a and b, and elliptic curve cofactor h.

Output: An elliptic curve point G or "failure".

Actions: Generate an elliptic curve point G as follows:

1. Let $A = 4261736520706F696E74_{16}$, which is the octet string associated with the ASCII representation of the text string "Base point".
2. Let $B = 01_{16}$, an octet string of length 1.
3. Let $c = 1$.
4. Convert integer c to an octet string C of length $1 + \lfloor \log_{256}(c) \rfloor$.
5. Let $H = Hash(A \| B \| C \| S)$.
6. Convert H to an integer e.
7. Let $t = e \bmod 2q$.
8. Let $u = t \bmod q$ and $z = \lfloor t/q \rfloor$.
9. Convert integer u to a field element $x \in \mathbb{F}_q$.
10. Recover a y-coordinate from the compressed point information (x, z), as appropriate to the elliptic curve.

11. If there is no valid y, then increment c and go back to Step 4.
12. Let $R = (x, y)$.
13. Compute $G = hR$.
14. Output G.

3.2 Elliptic Curve Key Pairs

All the public-key cryptographic schemes described in this document use key pairs known as elliptic curve key pairs.

Given some elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$, an elliptic curve key pair (d, Q) associated with T consists of an elliptic curve secret key d which is an integer in the interval $[1, n-1]$, and an elliptic curve public key $Q = (x_Q, y_Q)$ which is the point $Q = dG$.

Section 3.2.1 describes how to generate elliptic curve key pairs, Section 3.2.2 describes how to validate elliptic curve public keys, and Section 3.2.3 describes how to partially validate elliptic curve public keys.

3.2.1 Elliptic Curve Key Pair Generation Primitive

Elliptic curve key pairs should be generated as follows:

Input: Valid elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$.

Output: An elliptic curve key pair (d, Q) associated with T.

Actions: Generate an elliptic curve key pair as follows:

1. Randomly or pseudorandomly select an integer d in the interval $[1, n-1]$.
2. Compute $Q = dG$.
3. Output (d, Q).

3.2.2 Validation of Elliptic Curve Public Keys

Frequently, it is either necessary or desirable for an entity using an elliptic curve public key to receive an assurance that the public key is valid — that is, that it satisfies the arithmetic requirements of an elliptic curve public key — either to prevent malicious insertion of an invalid public key to enable attacks like small subgroup attacks, or to detect inadvertent coding or transmission errors.

There are four acceptable methods for an entity U to receive an assurance that an elliptic curve public key is valid. At least one of the methods must be supplied, although in many cases greater security may be obtained by carrying out more than one of the methods.

The four acceptable methods are:

1. Entity U performs validation of the elliptic curve public key itself using the public key validation primitive described in Section 3.2.2.1.

2. Entity U generates the elliptic curve public key itself using a trusted system that in turn uses the primitive specified in Section 3.2.1.

3. Entity U receives assurance in an authentic manner that a party trusted with respect to U's use of the elliptic curve public key has performed validation of the public key using the public key validation primitive described in Section 3.2.2.1.

4. Entity U receives assurance in an authentic manner that a party trusted with respect to U's use of the elliptic curve public key generated the public key using a trusted system that in turn uses the primitive specified in Section 3.2.1.

Usually, when U accepts another party's assurance that an elliptic curve public key is valid, the other party is a CA who validated the public key during the certification process. Occasionally, U may also receive assurance from another party other than a CA. For example, it may be acceptable for U to accept assurance from V that the public key is valid if the public key is received in a signed message, such as a message signed by V.

3.2.2.1 Elliptic Curve Public Key Validation Primitive

The elliptic curve public key validation primitive should be used to check that an elliptic curve public key is valid as follows:

Input: Valid elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$, and an elliptic curve public key $Q = (x_Q, y_Q)$ associated with T.

Output: An indication of whether the elliptic curve public key is valid or not — either "valid" or "invalid".

Actions: Validate the elliptic curve public key as follows:

1. Check that $Q \neq \mathcal{O}$.

2. If T represents elliptic curve domain parameters over $\mathbb{F}_p$, check that x_Q and y_Q are integers in the interval $[0, p-1]$, and that:

$$y_Q^2 \equiv x_Q^3 + ax_Q + b \pmod{p}.$$

3. If T represents elliptic curve domain parameters over $\mathbb{F}_{2^m}$, check that x_Q and y_Q are binary polynomials of degree at most $m-1$, and that:

$$y_Q^2 + x_Q y_Q = x_Q^3 + ax_Q^2 + b \text{ in } \mathbb{F}_{2^m}.$$

4. Check that $nQ = \mathcal{O}$.

5. If any of the checks fail, output "invalid", otherwise output "valid".

In the routine above, Steps 1, 2, and 3 check that Q is a point on E other than the point at infinity, and Step 4 checks that Q is a scalar multiple of G.

In Step 4, it may not be necessary to compute the point nQ. For example, if $h = 1$, then $nQ = \mathcal{O}$ is implied by the checks in Steps 2 and 3, because this property holds for all points $Q \in E$. If $h = 2$ and T represents elliptic curve domain parameters over $\mathbb{F}_{2^m}$, then it suffices to check that the trace of x_Q is 1. Similar checks may be performed in other situations where h is small.

3.2.3 Partial Validation of Elliptic Curve Public Keys

Sometimes it is sufficient for an entity using an elliptic curve public key to receive an assurance that the public key is partially valid, rather than "fully" valid — here an elliptic curve public key Q is said to be partially valid if Q is a point on the associated elliptic curve but it is not necessarily the case that $Q = dG$ for some d.

The MQV key agreement scheme and the Diffie-Hellman scheme using the cofactor Diffie-Hellman primitive are both examples of schemes designed to provide security even when entities only check that the public keys involved are partially valid. (This feature is desirable because it means that the schemes enjoy a computational advantage in some circumstances over schemes like the Diffie-Hellman scheme with the "standard" Diffie-Hellman primitive which require "fully" valid public keys. The computational advantage stems from the fact that public key partial validation is more efficient than public key "full" validation.)

There are four acceptable methods for an entity U to receive an assurance that an elliptic curve public key is partially valid. At least one of the methods must be supplied, although in many cases greater security may be obtained by carrying out more than one of the methods.

The four acceptable methods are:

1. Entity U performs partial validation of the elliptic curve public key itself using the public key partial validation primitive described in Section 3.2.3.1.

2. Entity U generates the elliptic curve public key itself using a trusted system that in turn uses the primitive specified in Section 3.2.1.

3. Entity U receives assurance in an authentic manner that a party trusted with respect to U's use of the elliptic curve public key has performed partial validation of the public key using the public key partial validation primitive described in Section 3.2.3.1.

4. Entity U receives assurance in an authentic manner that a party trusted with respect to U's use of the elliptic curve public key generated the public key using a trusted system that in turn uses the primitive specified in Section 3.2.1.

3.2.3.1 Elliptic Curve Public Key Partial Validation Primitive

The elliptic curve public key partial validation primitive should be used to check that an elliptic curve public key is partially valid as follows:

Input: Valid elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$, and an elliptic curve public key $Q = (x_Q, y_Q)$ associated with T.

Output: An indication of whether the elliptic curve public key is partially valid or not — either "valid" or "invalid".

Actions: Partially validate the elliptic curve public key as follows:

1. Check that $Q \neq \mathcal{O}$.
2. If T represents elliptic curve domain parameters over $\mathbb{F}_p$, check that x_Q and y_Q are integers in the interval $[0, p-1]$, and that:
$$y_Q^2 \equiv x_Q^3 + ax_Q + b \pmod{p}.$$
3. If T represents elliptic curve domain parameters over $\mathbb{F}_{2^m}$, check that x_Q and y_Q are binary polynomials of degree at most $m-1$, and that:
$$y_Q^2 + x_Q y_Q = x_Q^3 + ax_Q^2 + b \text{ in } \mathbb{F}_{2^m}.$$
4. If any of the checks fail, output "invalid", otherwise output "valid".

In the routine above, Steps 1, 2, and 3 check that Q is a point on E other than the point at infinity.

3.2.4 Verifiable and Assisted Key Pair Generation and Validation

In certain situations, an authority may wish to contribute to the generation of an entity U's key pair. For example, to be certain that entity U has not stolen or fabricated the key pair for a malicious purpose such as identity theft or an unknown key share attack, an authority may give some input into the key pair. As another example, if entity U is not able to provide sufficient entropy into the private key, the authority may wish to supplement the entropy while in a secure environment.

A self-signed signature is a signature in which the message signed contains the signature. It is possible to generate a self-signed ECDSA signature. This is done by selecting the signature first, then the rest of the message, and finally the key pair. Details for doing this are provided in Section 4.1.7.

In the case of ECDSA, a self-signed signature ensures a unique key pair per message signed. The function from a self-signed signature to a key pair is essentially one-way, so it is difficult to produce a self-signed signature that produces a target key pair.

If an authority contributes unique information to the message signed, then the authority ensures that the key pair is unique. A unique key pair ensures that the key pair is not another entity's key pair and that the key pair was not specifically created as part of an attack.

An authority can also contribute entropy to the key pair generation by providing some entropy for inclusion in the message to be signed. Inclusion of this information in the self-signed signature assures the authority that the entropy provided was employed in the key pair generation. A reason

to do this is if the key pair generator does not have very reliable entropy generation. In that case, the authority can assist the key pair generator. In this situation, however, to protect the security of the key pair, the authority must be trusted and the self-signed message must be kept confidential.

3.3 Elliptic Curve Diffie-Hellman Primitives

This section specifies the elliptic curve Diffie-Hellman primitives, which are the basis for the operation of the Elliptic Curve Integrated Encryption Scheme in Section 5.1, and the elliptic curve Diffie-Hellman scheme in Section 6.1.

Two primitives are specified: the elliptic curve Diffie-Hellman primitive and the elliptic curve cofactor Diffie-Hellman primitive. The basic idea of both primitives is the same — to generate a shared secret value from a private key owned by one entity U and a public key owned by another entity V so that if both entities execute the primitive simultaneously with corresponding keys as input they will recover the same shared secret value.

However the two primitives are subtly different: the elliptic curve Diffie-Hellman primitive is the straightforward analogue of the well-known Diffie-Hellman key agreement method, whereas the elliptic curve cofactor Diffie-Hellman primitive incorporates the cofactor into the calculation of the shared secret value to provide efficient resistance to attacks like small subgroup attacks.

The elliptic curve Diffie-Hellman primitive is specified in Section 3.3.1, and the elliptic curve cofactor Diffie-Hellman primitive is specified in Section 3.3.2.

3.3.1 Elliptic Curve Diffie-Hellman Primitive

Entity U should employ the following process to calculate a shared secret value with V using the elliptic curve Diffie-Hellman primitive:

Input: The elliptic curve Diffie-Hellman primitive takes as input:

1. Valid elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$.
2. An elliptic curve private key d_U associated with T owned by U.
3. A valid elliptic curve public key Q_V associated with T purportedly owned by V.

Output: A shared secret field element z, or "invalid".

Actions: Calculate a shared secret value as follows:

1. Compute the elliptic curve point $P = (x_P, y_P) = d_U Q_V$.
2. Check that $P \neq \mathcal{O}$. If $P = \mathcal{O}$, output "invalid" and stop.
3. Output $z = x_P$ as the shared secret field element.

3.3.2 Elliptic Curve Cofactor Diffie-Hellman Primitive

Entity U should employ the following process to calculate a shared secret value with V using the elliptic curve cofactor Diffie-Hellman primitive:

Input: The elliptic curve cofactor Diffie-Hellman primitive takes as input:

1. Valid elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$.
2. An elliptic curve private key d_U associated with T owned by U.
3. A partially valid elliptic curve public key Q_V associated with T purportedly owned by V.

Output: A shared secret field element z, or "invalid".

Actions: Calculate a shared secret value as follows:

1. Compute the elliptic curve point $P = (x_P, y_P) = hd_UQ_V$.
2. Check that $P \neq \mathcal{O}$. If $P = \mathcal{O}$, output "invalid" and stop.
3. Output $z = x_P$ as the shared secret field element.

3.4 Elliptic Curve MQV Primitive

This section specifies the elliptic curve MQV primitive, which is the basis for the operation of the elliptic curve MQV scheme specified in Section 6.2.

The basic idea of this primitive is to generate a shared secret value from two elliptic curve key pairs owned by one entity U and two elliptic curve public keys owned by another entity V so that if both entities execute the primitive simultaneously with corresponding keys as input they will recover the same shared secret value.

Entity U should employ the following process to calculate a shared secret value with entity V using the elliptic curve MQV primitive:

Input: The elliptic curve MQV primitive takes as input:

1. Valid elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$.
2. Two elliptic curve key pairs $(d_{1,U}, Q_{1,U})$ and $(d_{2,U}, Q_{2,U})$ associated with T owned by U.
3. Two partially valid elliptic curve public keys $Q_{1,V}$ and $Q_{2,V}$ associated with T purportedly owned by V.

Output: A shared secret field element z, or "invalid".

Actions: Calculate a shared secret value as follows:

1. Set $q = p$ if $T = (p, a, b, G, n, h)$, or $q = 2^m$ if $T = (m, f(x), a, b, G, n, h)$.

2. Compute an integer $\overline{Q_{2,U}}$ using $Q_{2,U} = (x_Q, y_Q)$ as follows:

   2.1. Convert x_Q to an integer x using the conversion routine specified in Section 2.3.9.

   2.2. Calculate:
   $$\overline{x} = x \bmod 2^{\lceil (\log_2 n)/2 \rceil}.$$

   2.3. Calculate:
   $$\overline{Q_{2,U}} = \overline{x} + 2^{\lceil (\log_2 n)/2 \rceil}.$$

3. Compute the integer:
   $$s = d_{2,U} + \overline{Q_{2,U}} d_{1,U} \bmod n.$$

4. Compute an integer $\overline{Q_{2,V}}$ using $Q_{2,V} = (x'_Q, y'_Q)$ as follows:

   4.1. Convert x'_Q to an integer x' using the conversion routine specified in Section 2.3.9.

   4.2. Calculate:
   $$\overline{x'} = x' \bmod 2^{\lceil (\log_2 n)/2 \rceil}.$$

   4.3. Calculate:
   $$\overline{Q_{2,V}} = \overline{x'} + 2^{\lceil (\log_2 n)/2 \rceil}.$$

5. Compute the elliptic curve point:
   $$P = (x_P, y_P) = hs(Q_{2,V} + \overline{Q_{2,V}} Q_{1,V}).$$

6. Check that $P \neq \mathcal{O}$. If $P = \mathcal{O}$, output "invalid" and stop.

7. Output $z = x_P$ as the shared secret field element.

3.5 Hash Functions

In 2005, an attack [WYY05b] was announced that finds a collision in SHA-1 in about 2^{69} hash operations. Subsequently, attacks using 2^{63} hash operations were announced [WYY05a].

These attacks decrease the security of SHA-1 against collision resistance. Collision resistance is primarily important for ECDSA in this standard, because it is necessary to resist existential forgery against ECDSA by an active chosen message attack.

In situations where only ECDSA with SHA-1 can be used, and 80 bits of security against existential forgery by active chosen message attacks is necessary, then one or more of the following countermeasures must be used:

- When verifying, obtain independent assurance that the signature was generated before the discovery of the attacks on SHA-1.

- When signing, ensure that the content of the message or part of the message is unpredictable enough that active chosen message attacks are infeasible. One way to do this is specified in [800-106].
- When signing or verifying, ensure that the form of the message is known not to be vulnerable to the existing collision attacks.

This section specifies the cryptographic hash functions supported in this document.

Hash functions are used by the verifiably random curve and base point generators specified in Section 3.1.3, by some of the key derivation functions specified in Section 3.6, by the HMAC message authentication code specified in Section 3.7, and by the ECDSA digital signature algorithm specified in Section 4.1.

The hash functions will be used to calculate the hash value associated with an octet string.

The list of supported hash functions at this time is:

SHA-1
SHA-224
SHA-256
SHA-384
SHA-512

These hash functions are specified in FIPS 180-2 [180-2]. They map octet strings of length less than a certain number octets to hash values which are octet strings of a fixed length.

NIST is holding a competition for a SHA-3 hash function, scheduled for completion in 2012. The SHA-3 hash function is to provide the same output lengths as SHA-2, but is intended to have less potential than SHA-2 of being vulnerable to possible extensions of the attacks on SHA-1. Future versions of this standard SEC 1 are likely to allow SHA-3, and perhaps other hashes, such as elliptic curve hash functions, if appropriate.

The security level associated with a hash function depends on its application. Where collision resistance is necessary, the security level is at most half the output length (in bits) of the hash function. Where collision resistance is not necessary, the security level is at most the length (in bits) of the hash function. Collision resistance is generally needed for computing message digests in ECDSA, but otherwise it may not be strictly necessary (such as, for use in key derivation, message authentication, random number generation and curve generation). Recent results have suggested that SHA-1 provides less than 80 bits of collision resistance. Therefore, SHA-1 should only be used for backwards compatibility when computing message digests with ECDSA. For other hash functions, the security level of collision resistance may be regarded as half the output length, until further notice. Therefore, for example, SHA-256 may be used to compute message digest for ECDSA at a security level of 128 bits. To promote interoperability, the choice of hash function for message digesting, message authentication (in HMAC as part of ECIES), for key derivation, and for curve generation, should be twice the desired security level. Exceptions may be made, primarily for efficiency or backwards interoperability reasons.

For clarity in the remainder of this section, the generic operation of hash functions is described so that their use can be precisely specified later on.

The values of *hashlen*, used below, for the hash functions SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 are, respectively, 20, 28, 32, 48, and 64. The value of *hashmaxlen*, used below, for the hash functions SHA-1, SHA-224, SHA-256 are $2^{61} - 1$, and the value of *hashmaxlen* for the hash functions SHA-384 and SHA-512 are $2^{125} - 1$.

Hash values should be calculated as follows:

Setup: Select one of the approved hash functions. Let *Hash* denote the hash function chosen, *hashlen* denote the length in octets of hash values computed using *Hash*, and *hashmaxlen* denote the maximum length in octets of messages that can be hashed using *Hash*.

Input: The input to the hash function is an octet string M.

Output: The hash value $H = Hash(M)$ which is an octet string of length *hashlen* octets, or "invalid".

Actions: Calculate the hash value H as follows:

1. Check that M is less than *hashmaxlen* octets long — i.e. check that:

   $$|M| < hashmaxlen.$$

   If $|M| \geq hashmaxlen$, output "invalid" and stop.

2. Convert M to a bit string $\overline{M}$ using the conversion routine specified in Section 2.3.2.

3. Calculate the hash value $\overline{H}$ corresponding to $\overline{M}$ using the selected hash function:

   $$\overline{H} = \overline{Hash}(\overline{M}).$$

   where $\overline{Hash}$ is the hash function algorithm as defined with bit string input and outputs.

4. Convert $\overline{H}$ to an octet string H using the conversion routine specified in Section 2.3.1.

5. Output H.

3.6 Key Derivation Functions

This section specifies the key derivation functions supported in this document.

The key derivation functions are used by encryption and key transport schemes specified in Section 5, and the key agreement schemes specified in Section 6.

The key derivation functions will be used to derive keying data from a shared secret octet string.

The list of supported key derivation functions at this time is:

ANSI-X9.63-KDF
IKEv2-KDF
TLS-KDF
NIST-800-56-Concatenation-KDF

The key derivation function ANSI-X9.63-KDF is the simple hash function construct described in ANS X9.63 [X9.63]. This key derivation function is described in Section 3.6.1.

The key derivation functions IKEv2-KDF and TLS-KDF shall only be used with the elliptic curve Diffie-Hellman scheme for use in the IKEv2 and TLS protocols, respectively. The function IKEv2-KDF is specified in [2409] and [4306]. The function TLS-KDF is specified in [2246] and [4492]. The function NIST-800-56-Concatenation-KDF is specified in [800-56A].

The NIST-800-56-Catenation-KDF should be used, except for backwards compatability with implementations already using one of the three other key derivation functions.

3.6.1 ANS X9.63 Key Derivation Function

Keying data should be calculated using ANSI-X9.63-KDF as follows:

Setup: Select one of the approved hash functions listed in Section 3.5. Let *Hash* denote the hash function chosen, *hashlen* denote the length in octets of hash values computed using *Hash*, and *hashmaxlen* denote the maximum length in octets of messages that can be hashed using *Hash*.

Input: The input to the key derivation function is:

1. An octet string Z which is the shared secret value.
2. An integer *keydatalen* which is the length in octets of the keying data to be generated.
3. (Optional) An octet string *SharedInfo* which consists of some data shared by the entities intended to share the shared secret value Z.

Output: The keying data K which is an octet string of length *keydatalen* octets, or "invalid".

Actions: Calculate the keying data K as follows:

1. Check that $|Z| + |SharedInfo| + 4 < hashmaxlen$. If $|Z| + |SharedInfo| + 4 \geq hashmaxlen$, output "invalid" and stop.
2. Check that $keydatalen < hashlen \times (2^{32} - 1)$. If $keydatalen \geq hashlen \times (2^{32} - 1)$, output "invalid" and stop.
3. Initiate a 4 octet, big-endian octet string *Counter* as 00000001_{16}.
4. For $i = 1$ to $\lceil keydatalen/hashlen \rceil$, do the following:
   4.1. Compute:
   $$K_i = Hash(Z \,\|\, Counter \,\|\, [SharedInfo])$$
   using the selected hash function from the list of approved hash functions in Section 3.5.
   4.2. Increment *Counter*.
   4.3. Increment i.

5. Set K to be the leftmost *keydatalen* octets of:

$$K_1 \| K_2 \| \ldots \| K_{\lceil keydatalen/hashlen \rceil}.$$

6. Output K.

3.7 MAC schemes

This section specifies the MAC schemes supported in this document.

The MAC schemes will be used by ECIES specified in Section 5.1.

MAC schemes are designed to be used by two entities — a sender U and a recipient V — when U wants to send a message M to V in an authentic manner and V wants to verify the authenticity of M.

Here the MAC schemes are described in terms of a tagging operation, a tag checking operation, and associated setup and key deployment procedures. Entities U and V should use the schemes as follows when they want to communicate. First, U and V should use the setup and key deployment procedures to establish which options to use the scheme with, and to create a shared secret key K to control the tagging and tag checking operations. Then each time U wants to send a message M to V, entity U should apply the tagging operation to M under the shared secret key K to compute the tag D on M, and convey M and D to V. Finally, when V receives M and D, entity V should apply the tag checking operation to M and D under K to verify the authenticity of M. If the tag checking operation outputs "valid", V concludes that M is indeed authentic.

Loosely speaking, MAC schemes are designed so that it is hard for an adversary to forge valid message and tag pairs. In other words, MAC schemes provide data origin authentication and data integrity.

The list of supported MAC schemes at this time is:

HMAC–SHA-1–160 with 20 octet or 160 bit keys
HMAC–SHA-1–80 with 20 octet or 160 bit keys
HMAC–SHA-224–112 with 28 octet or 224 bit keys
HMAC–SHA-224–224 with 28 octet or 224 bit keys
HMAC–SHA-256–128 with 32 octet or 256 bit keys
HMAC–SHA-256–256 with 32 octet or 256 bit keys
HMAC–SHA-384–192 with 48 octet or 384 bit keys
HMAC–SHA-384–284 with 48 octet or 384 bit keys
HMAC–SHA-512–256 with 64 octet or 512 bit keys
HMAC–SHA-512–512 with 64 octet or 512 bit keys
CMAC–AES–128
CMAC–AES–192
CMAC–AES–256

The first two of these MAC schemes are specified in [2104] and [X9.71] based on the hash function SHA-1 specified in [180-1]. The remaining HMAC schemes are introduced in [198], while the

CMAC schemes are introduced in [800-38B]. Following the notation suggested in [2104], here HMAC–*Hash*–x denotes the HMAC function used in conjunction with the hash function *Hash* to produce message tags of length $x/8$ octets or x bits. In the case of CMAC, the notation CMAC–AES–x denotes that the block cipher to be used is AES–x, where x is key size in bits. The tag length for CMAC–AES–x is always 128 bits. All the supported MAC schemes are designed to be existentially unforgeable in the presence of an adversary capable of launching chosen-message attacks.

(Note that this document does not suggest that other MAC schemes should not be used elsewhere in a system — it merely says that only the MAC schemes listed above should be used to build ECIES.)

For clarity in the remainder of this section, the generic operation of the MAC schemes by U and V is described so that the use of the schemes can be specified precisely later on. The setup procedure is described in Section 3.7.1, the key deployment procedure is specified in Section 3.7.2, the tagging operation is specified in Section 3.7.3, and the tag checking operation is specified in Section 3.7.4.

3.7.1 Scheme Setup

Entities U and V should perform the following setup procedure to use a MAC scheme:

1. Entities U and V should establish which of the supported MAC schemes to use (and if appropriate select any initial values required by the MAC scheme). Let *MAC* denote the MAC scheme chosen, *mackeylen* denote the length in octets of the keys used by the scheme, and *maclen* denote the length in octets of the tags produced by the scheme.

3.7.2 Key Deployment

Entities U and V should perform the following key deployment procedure to use the MAC scheme:

1. Entities U and V should establish a shared secret key K of length *mackeylen* octets. K should be chosen randomly or pseudorandomly.

3.7.3 Tagging Operation

Entity U should tag messages to send to V using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: An octet string M which is the data to be tagged.

Output: An octet string D of length *maclen* octets which is the tag on M, or "invalid".

Actions: Compute the tag D on M as follows:

1. Convert M to a bit string $\overline{M}$ and K to a bit string $\overline{K}$ using the conversion routine specified in Section 2.3.2.

2. Calculate the tag $\overline{D}$ on $\overline{M}$ using the selected MAC scheme under the shared secret key $\overline{K}$:

$$\overline{D} = MAC_{\overline{K}}(\overline{M}).$$

If the MAC scheme outputs "invalid", output "invalid" and stop.

3. Convert $\overline{D}$ to an octet string D using the conversion routine specified in Section 2.3.1

4. Output the octet string D of length *maclen* octets.

3.7.4 Tag Checking Operation

Entity V should check the authenticity of tagged messages from U using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: The input to the tag checking operation is:

1. An octet string M which is the message.

2. An octet string D which is the purported tag on M.

Output: An indication of whether the tagged message is valid or not — either "valid" or "invalid".

Actions: Check the tag D on M as follows:

1. Convert M to a bit string $\overline{M}$, D to a bit string $\overline{D}$, and K to a bit string $\overline{K}$ using the conversion routine specified in Section 2.3.2.

2. Calculate the tag $\overline{D'}$ on $\overline{M}$ using the selected MAC scheme under the shared secret key $\overline{K}$:

$$\overline{D'} = MAC_{\overline{K}}(\overline{M}).$$

If the MAC scheme outputs "invalid", output "invalid" and stop.

3. Compare $\overline{D'}$ and $\overline{D}$. If $\overline{D'} = \overline{D}$, output "valid", and if $\overline{D'} \neq \overline{D}$, output "invalid".

3.8 Symmetric Encryption Schemes

This section specifies the symmetric encryption schemes supported in this document.

The symmetric encryption schemes will be used by the Elliptic Curve Integrated Encryption Scheme (ECIES) specified in Section 5.1. It cannot be overemphasized that, unless otherwise stated, the discussions of the symmetric encryption schemes here apply to its use in ECIES but not necessarily to its use more general applications. In particular, ECIES as specified ensures that a single symmetric encryption key is never used to encrypt two different messages, because each time a message is encrypted using ECIES a new symmetric encryption key is derived.

Symmetric encryption schemes are designed to be used by two entities — a sender U and a recipient V — when U wants to send a message M to V confidentially, and V wants to recover M.

Here symmetric encryption schemes are described in terms of an encryption operation, a decryption operation, and associated setup and key deployment procedures. Entities U and V should use the scheme as follows when they want to communicate. First U and V should use the setup and key deployment procedures to establish which options to use the scheme with, and to create a shared secret key K to control the encryption and decryption operations. Then each time U wants to send a message M to V, U should apply the encryption operation to M under the shared secret key K to compute the encryption or ciphertext C of M, and convey C to V. Finally when V receives C, V should apply the decryption operation to C under K to recover the message M.

Loosely speaking, symmetric encryption schemes are designed so that it is hard for an adversary to recover messages from their ciphertexts. In other words, symmetric encryption schemes provide data confidentiality.

The list of supported symmetric encryption schemes at this time is:

3-key TDES in CBC mode
XOR encryption scheme
AES–128 in CBC mode
AES–192 in CBC mode
AES–256 in CBC mode
AES–128 in CTR mode
AES–192 in CTR mode
AES–256 in CTR mode

The block cipher 3-key TDES in CBC mode is specified in ANS X9.52 [X9.52]. Here it is considered to use shared secret keys of length 24 octets or 192 bits — which are split up into 3 subkeys K_1, K_2, and K_3 by interpreting the leftmost 8 octets or 64 bits as K_1, the middle 8 octets or 64 bits as K_2, and the rightmost 8 octets or 64 bits as K_3, and replacing the appropriate bits of K_1, K_2, and K_3 with parity bits.

The block cipher AES is specified in [197]. The CBC and CTR modes of AES are specified in [800-38A].

Furthermore here the 8 octet or 64 bit IV for TDES in CBC mode should always take the value 00000000_{16}. Furthermore here the 16 octet or 128 bit IV for AES in CBC mode should always take the value 00000000000000000_{16}. Likewise, the ICB for AES in CTR mode should take the value 00000000000000000_{16}. Neither IV nor ICB, when used in ECIES, should be transmitted as part of the ciphertext.

The XOR encryption scheme is the simple encryption scheme in which encryption consists of XORing the key and the message, and decryption consists of XORing the key and the ciphertext to recover the message. The XOR scheme is commonly used either with truly random keys when it is known as the "one-time pad", or with pseudorandom keys as a component in the construction of stream ciphers. The XOR encryption scheme uses keys which are the same length as the message to be encrypted or the ciphertext to be decrypted.

The block ciphers 3-key TDES and AES in CBC mode are designed to provide semantic security in the presence of adversaries launching chosen-message and chosen-ciphertext attacks. The XOR encryption scheme is designed to provide semantic security when used to encrypt a single message

in the presence of adversaries only capable of launching passive attacks. (Although this limits use of the XOR encryption scheme in general, it is sufficient for the purposes of building ECIES.)

The requirements above apply to ECIES. Other symmetric encryption schemes may be used elsewhere in the system. In general uses of CBC mode, the IV should be chosen as an unpredictable value. Likewise, in general use of CTR mode, the ICB should be selected securely.

For clarity in the remainder of this standard, the generic operation of the symmetric encryption schemes by U and V is described so that the use of the schemes can be specified precisely later on. The setup procedure is described in Section 3.8.1, the key deployment procedure is specified in Section 3.8.2, the encryption operation is specified in Section 3.8.3, and the decryption operation is specified in Section 3.8.4.

3.8.1 Scheme Setup

Entities U and V should perform the following setup procedure to use a symmetric encryption scheme:

1. Entities U and V should establish which of the supported symmetric encryption schemes to use (and if appropriate select any initial values required by the encryption scheme). Let *ENC* denote the encryption scheme chosen, and *enckeylen* denote the length in octets of the keys used by the scheme.

3.8.2 Key Deployment

Entities U and V should perform the following key deployment procedure to use the symmetric encryption scheme:

1. Entities U and V should establish a shared secret key K of length *enckeylen* octets.

3.8.3 Encryption Operation

Entity U should encrypt messages to send to entity V using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: An octet string M which is the data to be encrypted.

Output: An octet string C which is the ciphertext corresponding to M, or "invalid".

Actions: Compute the ciphertext C as follows:

1. Convert M to a bit string $\overline{M}$ and K to a bit string $\overline{K}$ using the conversion routine specified in Section 2.3.2.

2. Calculate the encryption $\overline{C}$ of $\overline{M}$ using the encryption operation of the selected symmetric encryption scheme under the shared secret key $\overline{K}$. If the encryption operation outputs "invalid", output "invalid" and stop.

3. Convert $\overline{C}$ to an octet string C using the conversion routine specified in Section 2.3.1.

4. Output the octet string C.

3.8.4 Decryption Operation

Entity V should decrypt ciphertext from entity U using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: An octet string C which is the ciphertext and a symmetric encryption key K.

Output: An octet string M which is the decryption of C, or "invalid".

Actions: Decrypt C as follows:

1. Convert C to a bit string $\overline{C}$ and K to a bit string $\overline{K}$ using the conversion routine specified in Section 2.3.2.

2. Calculate the decryption $\overline{M}$ of $\overline{C}$ using the decryption operation of the selected symmetric encryption scheme under the shared secret key $\overline{K}$. If the decryption operation outputs "invalid", output "invalid" and stop.

3. Convert $\overline{M}$ to an octet string M using the conversion routine specified in Section 2.3.1.

4. Output the octet string M.

3.9 Key Wrap Schemes

This subsection specifies that either the NIST AES key wrap algorithm or the CMS TDES key wrap algorithm

- must be used as the key wrap scheme in the Wrapped Key Transport Scheme, and
- should be used more generally when wrapping an existing symmetric key with another symmetric key.

The AES key wrap algorithm was first specified in [Nat01]. It has also been re-specified in [3394]. As of October 2007, ASC X9 is also re-specifying it, with some minor extension for additional input in [X9.102], and has requested public review of the algorithms therein [ASC04].

The AES key wrap algorithm may be used with the AES block cipher or the TDES block cipher. When using the AES block cipher to wrap keys, the AES key wrap algorithm must be used. When using the TDES block cipher, however, another key wrap algorithm, the CMS TDES key wrap algorithm, may be used for backwards interoperability reasons. This algorithm was first specified in [2630] and is also being re-specified in [X9.102].

For clarity in the remainder of this standard, the generic operation of the key wrap schemes by U and V is described so that the use of the schemes can be specified precisely later on. The setup procedure is described in Section 3.9.1, the key deployment procedure is specified in Section 3.9.2, the wrap operation is specified in Section 3.9.3, and the unwrap operation is specified in Section 3.9.4.

3.9.1 Key Wrap Scheme Setup

Entities U and V should perform the following setup procedure to use a key wrap scheme:

1. Entities U and V should establish which of the supported key wrap schemes to use (and if appropriate select any initial values required by the key wrap scheme). Let *WRAP* denote the encryption scheme chosen, and *wrapkeylen* denote the length in octets of the keys used by the scheme.

3.9.2 Key Wrap Schemes Key Generation

Entities U and V should perform the following key deployment procedure to use the key wrap scheme:

1. Entities U and V should establish a key-encryption key K of length *wrapkeylen* octets.

3.9.3 Key Wrap Schemes Wrap Operation

Entity U should wrap keys to send to entity V using the key-encryption key and key wrap parameters established during the setup procedure and the key deployment procedure as follows:

Input: A key-encryption key K and an octet string C which is the key to be wrapped.

Output: An octet string W which is the wrapped key corresponding to C, or "invalid".

Actions: Compute the wrapped key W as follows:

1. Convert C to a bit string $\overline{C}$ and K to a bit string $\overline{K}$ using the conversion routine specified in Section 2.3.2.
2. Calculate the wrapped key $\overline{W}$ of $\overline{C}$ using the key wrap operation of the selected key wrap scheme under the key-encryption key $\overline{K}$. If the key wrap operation outputs "invalid", output "invalid" and stop.
3. Convert $\overline{W}$ to an octet string W using the conversion routine specified in Section 2.3.1.
4. Output the octet string W.

3.9.4 Key Wrap Schemes Unwrap Operation

Entity V should unwrap a wrapped key from entity U using the key-encryption key and key wrap parameters established during the setup procedure and the key deployment procedure as follows:

Input: A key-encryption key K and an octet string W which is the wrapped key.

Output: An octet string C which is the unwrapping of W, or "invalid".

Actions: Unwrap W as follows:

1. Convert W to a bit string $\overline{W}$ and K to a bit string $\overline{K}$ using the conversion routine specified in Section 2.3.2.

2. Calculate the unwrapping $\overline{C}$ of $\overline{W}$ using the unwrap operation of the selected key wrap scheme under the shared key-encryption key $\overline{K}$. If the unwrap operation outputs "invalid", output "invalid" and stop.

3. Convert $\overline{C}$ to an octet string C using the conversion routine specified in Section 2.3.1.

4. Output the octet string C.

3.10 Random Number Generation

Cryptographic keys must be generated in a way that prevents an adversary from guessing the private key. Keys should be generated with the help of a random number generator.

Random number generators must comply with ANS X9.82 [X9.82] or corresponding NIST publication [800-90].

For completeness, one RNG is specified here.

Compliance to an RNG specification is not usually needed for interoperability. Nevertheless, because the secrecy of keys usually depends on the security of the RNG, compliance to a secure RNG specification is necessary. Failure to comply risks insecure key generation, which can undermine the security of an implementation.

3.10.1 Entropy

A random number generator (RNG) maintains a state. The output of the random number generator is a function of the state. The security of the RNG depends on the maximum probability that its state takes any one value. For a security level of t bits, the maximum probability of any state value must be at most 2^{-t}. Generally, the security level of a cryptographic system is no more than the security level of the RNG from which its cryptographic keys are derived.

When the maximum probability in a probability distribution is 2^{-t}, that distribution is said to have *min-entropy* of t bits. Min-entropy is never more than Shannon entropy. Shannon entropy is generally not enough to ensure adequate security in cryptography, because of pathological probability distributions. For example, an RNG producing bit strings of length 256 bits could have 128 bits of Shannon entropy but only 1 bit of min-entropy.

As a precautionary measure against the risk that two different RNGs will collide with the same state, an RNG should also be personalized with a value that is not likely to repeated. The personalization value need not be secret. See [800-90, §8.7.1].

3.10.2 Deterministic Generation of Pseudorandom Bit Strings

The output of an RNG must be a one-way transformation of its state to ensure that the state cannot be efficiently derived from the output. Several one-way functions are available.

The state should be updated with a one-way function, so that past states cannot be learnt from a future compromised state. This attribute is sometimes called forward secrecy or backtracking resistance. Some of the schemes in the standard, such as MQV, provide forward secrecy. When forward secrecy is a security objective of these schemes, then the RNG used must also provide forward secrecy.

In some circumstances, it is necessary that the RNG be able to recover from compromise of the current state. Such recovery can only be accomplished by the injection of new entropy. This security attribute is sometimes called recoverable security or prediction resistance. Prediction resistance is an optional attribute, for very high security applications.

3.10.2.1 Dual EC RNG

This section is intended to provide a re-specification of the "Dual EC DRBG" being specified in NIST Special Publication 800-90, *Recommendation for Random Number Generation Using Deterministic Bit Generators* and in Draft American National Standard X9.82, Part 3, *Deterministic Random Bit Generators Mechanisms*. Implementations of the "Dual EC DRBG" that comply with either of NIST SP 800-90 or ANS X9.82 will also comply with this standard. For completeness and convenience, the salient aspects of this random number generated are re-specified here, paraphrased in equivalent form.

A Dual EC RNG uses a set T of elliptic curve domain parameters and an extra parameter Q, a point on the elliptic curve. Another parameter is *outlen*, which is the number of bits per point. There are various operations involved in the Dual EC RNG, specified below.

The Dual EC RNG maintains a secret state, which is an integer s. The state must be initialized with sufficient entropy, and it must also be reseeded occasionally with additional entropy. Initialization and reseeding are not specified here at this time. For these operations, see the other specifications of the "Dual EC DRBG", such as NIST SP 800-90 and ANS X9.82-2.

Elliptic points over binary fields in the Dual EC RNG are represented using the canonical polynomial basis representation, as specified in Section 2.1.2.

3.10.2.1.1 Output Block This is an internal operation of the Dual EC RNG. Compute $(x_1, y_1) = sP$ and $(x_2, y_2) = sQ$. Convert x_1 to an integer s', which becomes the new state. Convert x_2 to a bit string b. Take the rightmost *outlen* bits of b to obtain a shorter bit string r (dropping some number of leftmost bits). If the elliptic curve is defined over the finite field $\mathbb{F}_{2^m}$ with $m = 409$, then drop the rightmost bit of r. The bit string r is the output block.

3.10.2.1.2 Output Bit String This is the main external operation of the Dual EC RNG. To generate a bit string of k bits, obtain successive output blocks $r_0, \ldots, r_j$ until their combined length exceeds k. Concatenate the blocks and output the leftmost k bits.

3.10.3 Converting Random Bit Strings to Random Numbers

Elliptic curve private keys are integers in a certain range. For full security, these integers should have a probability distribution that is as close as possible to uniform. (Otherwise, a variety of attacks may become possible.)

The deterministic algorithms in the previous section produce random bit strings. Bit strings can be converted to integers, but the range is not exactly that needed for the elliptic curve private keys. One of the following process may be used to convert a random bit string to a random integer, in such a way that if the bit string is uniform then so is the integer.

One of the methods [800-90, §B.5.1] from NIST Special Publication 800-90 on deterministic random number generation may be used.

The following alternative method may also be used. To generate a random integer k in the interval $[1, n-1]$, choose a random bit string B of a fixed length $m \geq \lceil \log_2(n-1) \rceil$. Convert B to an integer b, which will be in the interval $[0, 2^m - 1]$. Let $q = \lfloor 2^m/(n-1) \rfloor$. If $b < q(n-1)$, then let $k = 1 + (b \bmod n - 1)$. If $b \geq q(n-1)$, then select another B.

3.11 Security Levels and Protection Lifetimes

Data protected with cryptography today may continue to need protection in the future. Advances in cryptanalysis can be predicted, at least approximately.

Based on current approximations, this document requires that data that needs protection beyond the year 2010 must be protected with 112-bit security or higher. Data that needs protection beyond the year 2030 must be protected with 128-bit security or higher.

Data that needs protection beyond the year 2040 should be protected with 192-bit security or higher. Data that needs protection beyond 2080 should be protected with 256-bit security or higher.

4 Signature Schemes

This section specifies the signature schemes based on ECC supported in this document.

Signature schemes are designed to be used by two entities — a signer U and a verifier V — when U wants to send a message M in an authentic manner and V wants to verify the authenticity of M. In fact, once a message is signed, any entity V having a copy of U's public key may verify the signature. In particular, the verifier may not be the entity to whom U originally sent the message. Such third party verification is important for non-repudiation.

Here, signature schemes are described in terms of a signing operation, a verifying operation, and associated setup and key deployment procedures. Entities U and V should use the schemes as follows when they want to communicate. First U and V should use the setup procedure to establish which options to use the scheme with, then U should use the key deployment procedure to select a key pair and V should obtain U's public key — U will use the key pair to control the signing operation, and V will use the public key to control the verifying operation. Then, each time U wants to send a message M, entity U should apply the signing operation to M under its key pair to obtain a signature S on M, form a signed message from M and S, and convey the signed message to V. Finally, when V receives the signed message, entity V should apply the verifying operation to the signed message under U's public key to verify its authenticity. If the verifying operation outputs "valid", entity V concludes the signed message is indeed authentic.

There are two types of signature schemes, depending on the form of the signed message U must convey to V: signature schemes with appendix in which U must convey both M and S to V, and signature schemes with message recovery in which M can be recovered from S, so U need convey only S to V.

Loosely speaking, signature schemes are designed so that it is hard for an adversary who does not know U's secret key to forge valid signed messages. Thereby, signature schemes provide data origin authentication, data integrity, and non-repudiation.

The only signature scheme supported at this time is the Elliptic Curve Digital Signature Algorithm (ECDSA). ECDSA is specified in Section 4.1.

See Appendix B for a commentary on the contents on this section, including implementation discussion, security discussion, and references.

4.1 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a signature scheme with appendix based on ECC. It is designed to be existentially unforgeable, even in the presence of an adversary capable of launching chosen-message attacks.

The setup procedure for ECDSA is specified in Section 4.1.1, the key deployment procedure is specified in Section 4.1.2, the signing operation is specified in Section 4.1.3, and the verifying operation is specified in Section 4.1.4.

4.1.1 Scheme Setup

Entities U and V must perform the following setup procedure to prepare to use ECDSA:

1. Entity U should establish which of the hash functions supported in Section 3.5 to use when generating signatures. Let *Hash* denote the hash function chosen, and *hashlen* denote the length in octets of the hash values produced using *Hash*.
2. Entity U should establish elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $T = (m, f(x), a, b, G, n, h)$ at the desired security level. The elliptic curve domain parameters T should be generated using the primitive specified in Section 3.1.1.1 or the primitive specified in Section 3.1.2.1. Entity U should receive an assurance that the elliptic curve domain parameters T are valid using one of the methods specified in Section 3.1.1.2 or Section 3.1.2.2.
3. Entity V should obtain in an authentic manner the hash function *Hash* and elliptic curve domain parameters T established by U.

Entity V must receive an assurance that the elliptic curve domain parameters T are valid using one of the methods specified in Section 3.1.1.2 or Section 3.1.2.2.

4.1.2 Key Deployment

Entities U and V must perform the following key deployment procedure to prepare to use ECDSA:

1. Entity U should establish an elliptic curve key pair (d_U, Q_U) associated with T to use with the signature scheme. The key pair should be generated using the primitive specified in Section 3.2.1.
2. Entity V should obtain in an authentic manner the elliptic curve public key Q_U selected by U.

Entity V must receive an assurance that the elliptic curve public key Q_U is valid using one of the methods specified in Section 3.2.2.

4.1.3 Signing Operation

Entity U must sign messages using ECDSA using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: The signing operation takes as input an octet string M which is the message to be signed.

Output: A signature $S = (r, s)$ on M consisting of a pair of integers r and s, or "invalid".

Actions: Generate a signature S on M as follows:

1. Select an ephemeral elliptic curve key pair (k, R) with $R = (x_R, y_R)$ associated with the elliptic curve domain parameters T established during the setup procedure using the key pair generation primitive specified in Section 3.2.1.

2. Convert the field element x_R to an integer $\overline{x_R}$ using the conversion routine specified in Section 2.3.9.

3. Set $r = \overline{x_R} \bmod n$. If $r = 0$, or optionally r fails to meet other publicly verifiable criteria (see below), return to Step 1.

4. Use the hash function selected during the setup procedure to compute the hash value:

$$H = Hash(M)$$

of length *hashlen* octets as specified in Section 3.5. If the hash function outputs "invalid", output "invalid" and stop.

5. Derive an integer e from H as follows:

   5.1. Convert the octet string H to a bit string $\overline{H}$ using the conversion routine specified in Section 2.3.2.

   5.2. Set $\overline{E} = \overline{H}$ if $\lceil \log_2 n \rceil \geq 8(hashlen)$, and set $\overline{E}$ equal to the leftmost $\lceil \log_2 n \rceil$ bits of $\overline{H}$ if $\lceil \log_2 n \rceil < 8(hashlen)$.

   5.3. Convert the bit string $\overline{E}$ to an octet string E using the conversion routine specified in Section 2.3.1.

   5.4. Convert the octet string E to an integer e using the conversion routine specified in Section 2.3.8.

6. Compute:

$$s = k^{-1}(e + rd_U) \bmod n.$$

If $s = 0$, return to Step 1.

7. Output $S = (r, s)$. Optionally, output additional information needed to recover R efficiently from r (see below).

The signer may replace (r, s) with $(r, -s \mod n)$, because this is an equivalent signature.

The publicly verifiable criteria that r may be conditioned to satisfy may include that x_R is uniquely recoverable from r in that only one of the integers $\overline{x_R} = r + jn$ for $j \in \{0, 1, 2 \ldots, h\}$ represents a valid x-coordinate of a multiple of G. For the recommended curves [SEC 2] with $h = 1$ and $h = 2$, the number of valid candidate x-coordinates is usually one, so this is a vacuous check.

The additional information needed to compute R can consist of the point R itself, in either compressed or uncompressed form. However, since r provides considerable information about x_R, it is often sufficient to provide no extra information to determine x_R. At worst, $\log_2(h + 1)$ bits are needed to find x_R from r. In any case, information needed to recover y_R can take the form of single bit, or the full value of y_R depending on whether compactness or speed is preferred.

4.1.4 Verifying Operation

Entity V must verify signed messages from entity U using ECDSA using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: The verifying operation takes as input:

1. An octet string M which is the message.
2. Entity U's purported signature $S = (r, s)$ on M.
3. Optional: extra information to recover R efficiently from r (see below).

Output: An indication of whether the purported signature on M is valid or not — either "valid" or "invalid".

Actions: Verify the purported signature S on M as follows:

1. If r and s are not both integers in the interval $[1, n-1]$, output "invalid" and stop.
2. Use the hash function established during the setup procedure to compute the hash value:

   $$H = Hash(M)$$

   of length *hashlen* octets as specified in Section 3.5. If the hash function outputs "invalid", output "invalid" and stop.
3. Derive an integer e from H as follows:
   3.1. Convert the octet string H to a bit string $\overline{H}$ using the conversion routine specified in Section 2.3.2.
   3.2. Set $\overline{E} = \overline{H}$ if $\lceil \log_2 n \rceil \geq 8(hashlen)$, and set $\overline{E}$ equal to the leftmost $\lceil \log_2 n \rceil$ bits of $\overline{H}$ if $\lceil \log_2 n \rceil < 8(hashlen)$.
   3.3. Convert the bit string $\overline{E}$ to an octet string E using the conversion routine specified in Section 2.3.1.
   3.4. Convert the octet string E to an integer e using the conversion routine specified in Section 2.3.8.
4. Compute:

   $$u_1 = es^{-1} \bmod n \quad \text{and} \quad u_2 = rs^{-1} \bmod n.$$

5. Compute:

   $$R = (x_R, y_R) = u_1 G + u_2 Q_U.$$

   If $R = \mathcal{O}$, output "invalid" and stop.
6. Convert the field element x_R to an integer $\overline{x_R}$ using the conversion routine specified in Section 2.3.9.

7. Set $v = \overline{x_R} \bmod n$.

8. Compare v and r — if $v = r$, output "valid", and if $v \neq r$, output "invalid".

The optional input of extra information used to recover R efficiently from r is not used in the actions above. Nevertheless, it may be used in an equivalent sequence of actions to achieve more efficient verification. For example, if one recovers R from r, then one may verify that $sR = eG + rQ_U$. More generally, one may choose some integer t, and verify that $tsR = teG + trQ_U$. A t can be chosen so that both the integers $(ts \bmod n)$ and $(tr \bmod n)$ have size approximately $\sqrt{n}$, which can be used to make the verification operation faster.

4.1.5 Alternative Verifying Operation

A signer U may verify U's own signatures more efficiently with the following operation, which uses U's own private key.

A situation where this could be useful is when a CA verifies its own certificates.

All verification steps are the same, except that in Step 5, the verifier instead computes

$$R = (x_R, y_R) = (u_1 + u_2 d)G$$

The benefits of this are that the verifier needs just a single scalar multiplication, and pre-computed multiples of G can accelerate this computation.

4.1.6 Public Key Recovery Operation

Given an ECDSA signature (r, s) and EC domain parameters, it is generally possible to determine the public key Q, at least to within a small number of choices.

This is useful for generating self-signed signatures.

This is also useful in bandwidth constrained environments, when transmission of public keys cannot be afforded. Entity U could send a signature to entity V, who recovers Q_U. Entity V can look up the public key in some certificate or directory, and if it matches then the signature can be accepted. Alternatively, entity U may transmit the signature together with the certificate except that the public key is omitted from the certificate. For example, in long certificate chains signed with ECDSA, bandwidth can be saved by omission of the public keys.

Potentially, several candidate public keys can be recovered from a signature. At a small cost, the signer can generate the ECDSA signature in such a way that only one of the candidate public keys is viable, and such that the verifier has a very small additional cost of determining which is the correct public key.

Input: The public key recovery operations takes as input:

1. Elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $T = (m, f(x), a, b, G, n, h)$ at the desired security level.

2. A message M.

3. An ECDSA signature value (r, s) that is valid on message M for some public key to be determined.

Output: An elliptic curve public key Q for which (r, s) is a valid signature on message M.

Actions: Find public key Q as follows.

1. For j from 0 to h do the following.

   1.1. Let $x = r + jn$.

   1.2. Convert the integer x to an octet string X of length $mlen$ using the conversion routine specified in Section 2.3.7, where $mlen = \lceil (\log_2 p)/8 \rceil$ or $mlen = \lceil m/8 \rceil$.

   1.3. Convert the octet string $02_{16} \| X$ to an elliptic curve point R using the conversion routine specified in Section 2.3.4. If this conversion routine outputs "invalid", then do another iteration of Step 1.

   1.4. If $nR \neq \mathcal{O}$, then do another iteration of Step 1.

   1.5. Compute e from M using Steps 2 and 3 of ECDSA signature verification.

   1.6. For k from 1 to 2 do the following.

      1.6.1. Compute a candidate public key as:

$$Q = r^{-1}(sR - eG).$$

      1.6.2. Verify that Q is the authentic public key. (For example, verify the signature of a certification authority in a certificate which has been truncated by the omission of Q from the certificate.) If Q is authenticated, stop and output Q.

      1.6.3. Change R to $-R$.

2. Output "invalid".

4.1.7 Self-Signing Operation

Self-signed ECDSA signatures are useful for verfiable key generation, as described in Section 3.2.4.

To generate a self-signed ECDSA signature, the following operation can be used.

Input: The self-signing operation takes as input:

1. Elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $T = (m, f(x), a, b, G, n, h)$ at the desired security level.

2. Information I to be incorporated in the self-signed signature. Information I should include the identity of the signer.

Output: The self-signing operation generation produces as output:

1. An elliptic curve key pair (d, Q) associated with the domain parameters T.
2. A message M, which contains a copy of information I and a copy of a valid ECDSA signature (r, s) on message M under public key Q. Because (r, s) is a signature on itself plus other information, it is a self-signed signature.

Actions: Generate a self-signed signature message M and elliptic curve key pair as follows:

1. Select an ephemeral key pair (k, R) associated with the selected elliptic curve domain parameters.
2. Compute r from R as in Steps 1, 2, and 3 of ECDSA signature generation.
3. Select a random integer s in the interval $[1, n-1]$.
4. Form a message M containing both I and (r, s).
5. Use the Public Key Recovery Operation in Section 4.1.6, with inputs T, M and (r, s), to recover a public key Q.
6. Compute the private key d as follows:

$$d = r^{-1}(sk - e) \bmod n.$$

Another entity V, given M, can extract the signature (r, s) from M. Then V can verify the signature using the normal ECDSA verification operation.

If the signer can only generate an ephemeral private key k with entropy lower than necessary for the desired security level, then a trusted authority may supply supplemental entropy inside I. The trusted authority can then verify that the information I was indeed used to generate the key pair. In this case, the self-signed M must be kept secret from untrusted entities.

5 Encryption and Key Transport Schemes

This section specifies the public-key encryption and key transport schemes based on ECC supported in this document.

Public-key encryption schemes are designed to be used by two entities — a sender U and a recipient V — when U wants to send a message M to V confidentially, and V wants to recover M.

Key transport schemes are a special class of public-key encryption schemes where the message M is restricted to be a cryptographic key, usually a symmetric key. Except for this restriction, most of the discussion below about public-key encryption schemes also applies to key transport schemes.

Here, public-key encryption schemes are described in terms of an encryption operation, a decryption operation, and associated setup and key deployment procedures. Entities U and V should use the scheme as follows when they want to communicate. First U and V should use the setup procedure to establish which options to use the scheme with, then V should use the key deployment procedure to select a key pair and U should obtain V's public key — U will use V's public key to control the encryption procedure, and V will use its key pair to control the decryption operation. Then each time U wants to send a message M to V, U should apply the encryption operation to M under V's public key to compute an encryption or ciphertext C of M, and convey C to V. Finally when V receives C, entity V should apply the decryption operation to C under its key pair to recover the message M.

Loosely speaking, public-key encryption schemes are designed so that it is hard for an adversary who does not possess V's secret key to recover messages from their ciphertexts. Thereby, public-key encryption schemes provide data confidentiality.

The public-key encryption schemes specified in this section may be used to encrypt messages of any kind. They may be used to transport keying data from U to V, or to encrypt information data directly. This flexibility allows the schemes to be applied in a broad range of cryptographic systems. Nonetheless, it is envisioned that the majority of applications will apply the schemes for key transport, and subsequently use the transported key in conjunction with a symmetric bulk encryption scheme to encrypt information data. This is the traditional usage for public-key encryption schemes.

The public-key encryption schemes supported are the Elliptic Curve Integrated Encryption Scheme (ECIES) and the general construction of combining a key agreement scheme with a key wrap mechanism. The first, ECIES, is specified in Section 5.1. The second general construction is specified in Section 5.2.

See Appendix B for a commentary on the contents on this section, including implementation discussion, security discussion, and references.

5.1 Elliptic Curve Integrated Encryption Scheme

The Elliptic Curve Integrated Encryption Scheme (ECIES) is a public-key encryption scheme based on ECC. It is designed to be semantically secure in the presence of an adversary capable of launching chosen-plaintext and chosen-ciphertext attacks.

The setup procedure for ECIES is specified in Section 5.1.1, the key deployment procedure is specified in Section 5.1.2, the encryption operation is specified in Section 5.1.3, and the decryption operation is specified in Section 5.1.4.

5.1.1 Scheme Setup

Entities U and V should perform the following setup procedure to prepare to use ECIES:

1. Entity V should establish which of the key derivation functions supported in Section 3.6 to use, and select any options involved in the operation of the key derivation function. Let *KDF* denote the key derivation function chosen.

2. Entity V should establish which of the MAC schemes supported in Section 3.7 to use, and select any options involved in the operation of the MAC scheme. Let MAC denote the MAC scheme chosen, *mackeylen* denote the length in octets of the keys used by MAC, and *maclen* denote the length in octets of tags produced by MAC.

3. Entity V should establish which of the symmetric encryption schemes supported in Section 3.8 to use, and select any options involved in the operation of the encryption scheme. Let ENC denote the encryption scheme chosen, and *enckeylen* denote the length in octets of the keys used by ENC.

4. Entity V should establish whether to use the elliptic curve Diffie-Hellman primitive specified in Section 3.3.1, or the elliptic curve cofactor Diffie-Hellman primitive specified in Section 3.3.2.

5. Entity V should establish EC domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$ at the desired security level. The elliptic curve domain parameters T should be generated using the primitive specified in Section 3.1.1.1 or the primitive specified in Section 3.1.2.1. Entity V should receive an assurance that the elliptic curve domain parameters T are valid using one of the methods specified in Section 3.1.1.2 or Section 3.1.2.2.

6. Entity U should obtain in an authentic manner the selections made by V — the key derivation function *KDF*, the MAC scheme MAC, the symmetric encryption scheme ENC, the elliptic curve domain parameters T, and an indication whether to use the elliptic curve Diffie-Hellman primitive or the cofactor Diffie-Hellman. Entity U should also receive an assurance that the elliptic curve domain parameters T are valid using one of the methods specified in Section 3.1.1.2 or Section 3.1.2.2.

7. Entity U or V should establish whether or not to represent elliptic curve points using point compression.

8. Entities U and V should establish an expected format of $SharedInfo_2$ such that $EM \parallel SharedInfo_2$ can be uniquely parsed, which may be done if $SharedInfo_2$ is suffix-free, meaning that two different validly formatted values of $SharedInfo_2$ cannot be such that one is a suffix (i.e. tail) of the other. For example, this would be generally the case if $SharedInfo_2$ ended in a counter giving its length.

9. If the XOR symmetric encryption option is selected, then entities U and V should establish whether the backwards compatibility mode (that is, compatibility with version 1.0 of this standard) is desired.

5.1.2 Key Deployment

Entities U and V should perform the following key deployment procedure to prepare to use ECIES:

1. Entity V should establish an elliptic curve key pair (d_V, Q_V) associated with the elliptic curve domain parameters T established during the setup procedure. The key pair should be generated using the primitive specified in Section 3.2.1.
2. Entity U should obtain in an authentic manner the elliptic curve public key Q_V selected by V. If the elliptic curve Diffie-Hellman primitive is being used, U should receive an assurance that Q_V is valid using one of the methods specified in Section 3.2.2, and if the elliptic curve cofactor Diffie-Hellman primitive is being used, U should receive an assurance that Q_V is at least partially valid using one of the methods specified in Section 3.2.2 or Section 3.2.3.

5.1.3 Encryption Operation

Entity U should encrypt messages using ECIES using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: The input to the encryption operation is:

1. An octet string M which is the message to be encrypted.
2. (Optional) Two octet strings $\mathit{SharedInfo}_1$ and $\mathit{SharedInfo}_2$ which consist of some data shared by U and V.

Output: An octet string C which is an encryption of M, or "invalid".

Actions: Encrypt M as follows:

1. Select an ephemeral elliptic curve key pair (k, R) with $R = (x_R, y_R)$ associated with the elliptic curve domain parameters T established during the setup procedure. Generate the key pair using the key pair generation primitive specified in Section 3.2.1.
2. Decide whether or not to represent R using point compression according to the convention established during the setup procedure. Convert R to an octet string $\overline{R}$ using the conversion routine specified in Section 2.3.3.
3. Decide whether to use the elliptic curve Diffie-Hellman primitive or the elliptic curve cofactor Diffie-Hellman primitive according to the convention established during the setup procedure. Use the chosen Diffie-Hellman primitive specified in Section 3.3 to derive a shared secret field element $z \in \mathbb{F}_q$ from the ephemeral secret key k and V's public key Q_V obtained during the key deployment procedure. If the Diffie-Hellman primitive outputs "invalid", output "invalid" and stop.

4. Convert $z \in \mathbb{F}_q$ to an octet string Z using the conversion routine specified in Section 2.3.5.

5. Use the key derivation function *KDF* established during the setup procedure to generate keying data K of length *enckeylen* + *mackeylen* octets from Z and [*SharedInfo*$_1$]. If the key derivation function outputs "invalid", output "invalid" and stop.

6. Parse the leftmost *enckeylen* octets of K as an encryption key EK and the rightmost *mackeylen* octets of K as a MAC key MK. If symmetric encryption method is XOR and backwards compatibility mode is not selected, then instead parse the rightmost *enckeylen* octets of K as an encryption key EK and the leftmost *mackeylen* octets of K as a MAC key MK.

7. Use the encryption operation of the symmetric encryption scheme ENC established during the setup procedure to encrypt M under EK as ciphertext EM. If the encryption scheme outputs "invalid", output "invalid" and stop.

8. Use the tagging operation of the MAC scheme MAC established during the setup procedure to compute the tag D on $EM \,\|\, [\mathit{SharedInfo}_2]$ under MK. If the MAC scheme outputs "invalid", output "invalid" and stop.

9. Output $C = (\overline{R}, EM, D)$. Optionally, the ciphertext may be output as $C = \overline{R} \,\|\, EM \,\|\, D$.

5.1.4 Decryption Operation

Entity V should decrypt ciphertext using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: The input to the decryption operation is:

1. A triple of octet strings $C = (\overline{R}, EM, D)$ or an octet string C, which is the ciphertext.

2. (Optional) Two octet strings *SharedInfo*$_1$ and *SharedInfo*$_2$ which consist of some data shared by U and V.

Output: An octet string M which is the decryption of C, or "invalid".

Actions: Decrypt C as follows:

1. If C is an octet string and the leftmost octet of C is 02_{16} or 03_{16}, parse the leftmost $\lceil (\log_2 q)/8 \rceil + 1$ octets of C as an octet string $\overline{R}$, the rightmost *maclen* octets of C as an octet string D, and the remaining octets of C as an octet string EM. If the leftmost octet of C is 04_{16}, parse the leftmost $2\lceil (\log_2 q)/8 \rceil + 1$ octets of C as an octet string $\overline{R}$, the rightmost *maclen* octets of C as an octet string D, and the remaining octets of C as an octet string EM. If the leftmost octet of C is not 02_{16}, 03_{16}, or 04_{16}, output "invalid" and stop.

2. Convert the octet string $\overline{R}$ to an elliptic curve point $R = (x_R, y_R)$ associated with the elliptic curve domain parameters T established during the setup procedure using the conversion routine specified in Section 2.3.4. If the conversion routine outputs "invalid", output "invalid" and stop.

3. If the elliptic curve Diffie-Hellman primitive is being used, receive an assurance that R is a valid elliptic curve public key using one of the methods specified in Section 3.2.2. If the elliptic curve cofactor Diffie-Hellman primitive is being used, receive an assurance that R is at least a partially valid elliptic curve public key using one of the methods specified in Section 3.2.2 or Section 3.2.3. If an appropriate assurance is not obtained, output "invalid" and stop.

4. Decide whether to use the elliptic curve Diffie-Hellman primitive or the elliptic curve cofactor Diffie-Hellman primitive according to the convention established during the setup procedure. Use the chosen Diffie-Hellman primitive specified in Section 3.3 to derive a shared secret field element $z \in \mathbb{F}_q$ from V's secret key d_V established during the key deployment procedure and the public key R. If the Diffie-Hellman primitive outputs "invalid", output "invalid" and stop.

5. Convert $z \in \mathbb{F}_q$ to an octet string Z using the conversion routine specified in Section 2.3.5.

6. Use the key derivation function *KDF* established during the setup procedure to generate keying data K of length *enckeylen* + *mackeylen* octets from Z and $[\mathit{SharedInfo}_1]$. If the key derivation function outputs "invalid", output "invalid" and stop.

7. Parse the leftmost *enckeylen* octets of K as an encryption key EK and the rightmost *mackeylen* octets of K as a MAC key MK, with the following exception: if the symmetric encryption method is XOR and backwards compatibility mode is not selected, then instead parse the rightmost *enckeylen* octets of K as an encryption key EK and the leftmost *mackeylen* octets of K as a MAC key MK.

8. Use the tag checking operation of the MAC scheme *MAC* established during the setup procedure to check that D is the tag on $EM \,\|\, [\mathit{SharedInfo}_2]$ under MK. If the MAC scheme outputs "invalid", output "invalid" and stop.

9. Use the decryption operation of the symmetric encryption scheme *ENC* established during the setup procedure to decrypt EM under EK as M. If the encryption scheme outputs "invalid", output "invalid" and stop.

10. Output M.

5.2 Wrapped Key Transport Scheme

The wrapped key transport scheme uses a combination of a key wrap scheme and a key agreement scheme. The key agreement used can be either Diffie-Hellman (Section 6.1) or MQV (Section 6.2), but in either case it must be a 1-pass variant. In a 1-pass variant of a key agreement scheme, in the key deployment phase, entity U must obtain authentic copies of all of the keys of V, in addition to the usual key deployment operations. For Diffie-Hellman, entity U must obtain Q_V in an authentic manner. For MQV, entity U must obtain $Q_{2,V}$ in an authentic manner, which may be achieved most easily if $Q_{2,V} = Q_{1,V}$, which is the default choice in the absence of any indication otherwise.

Once U has obtained all of the keys of V in an authentic manner — for example, by extracting them from a certificate — then to complete key deployment, entity U needs only to send its own keys to entity V. Hence, all ephemeral keys are exchanged in a single pass.

In wrapped key transport, entity U uses a 1-pass key agreement operation with entity V to agree on a key K and then wraps a key C with K to obtain a wrapped key W. The wrapped key W is sent together with the public keys of U in the single pass of the exchange.

Any format for combining the public keys and wrapped keys into a single pass message is allowed, including the formats used in S/MIME [3278]. For convenience, this standard will include a pre-defined format that may find use in future applications.

A typical application of the transport key C is for encrypting or authenticating, or both encrypting and authenticating content data, in a single pass. The key C is often called a content-encryption key. Generally, the encrypted message and authentication tag, or both, will be sent in a single pass together with wrapped key and any necessary public keys. The most familiar single pass application is email.

If entity U wraps a single key C for many different recipients, which is useful for protecting an email sent to many recipients, say $V_1, V_2, \ldots$, then U may re-use the same ephemeral public key for each V_i. We denote by K_i the key agreed between U and V_i, and W_i the wrapping of C with K_i.

In this situation, entity V_i learns the key C. This brings a risk that entity V_i will abuse C by altering the message intended for V_j, thereby making V_j believe that the altered message came from U.

Entity U might wish to prevent this problem. If M is the message authenticated and $T = MAC_C(M)$ is the MAC tag, then entity U may include T as part of the *SharedInfo* used in the KDF with key agreement operation. When this is done, any modification to the message will modify T, which will modify K_j, which will modify W_j. Entity V_i will not be able to produce the correctly modified W_j, so if entity V_i modifies the message, then entity V_j will not re-compute the key C correctly and the message authentication will fail.

Alternatively, entity U may include T as an optional parameter into the key wrap scheme. This has a similar effect. Another option is for entity U to compute a separate tag T_i for each recipient. If the message is very long and the number of recipients is large, however, then computing many MAC tags on a long message is very slow. For a faster approach, entity U can compute $T_i = MAC_{K_i}(T)$, where T is as before.

6 Key Agreement Schemes

This section specifies the key agreement schemes based on ECC supported in this document.

Key agreement schemes are designed to be used by two entities — an entity U and an entity V — when U and V want to establish shared keying data that they can later use to control the operation of a symmetric cryptographic scheme.

Here, key agreement schemes are described in terms of a key agreement operation, and associated setup and key deployment procedures. Entities U and V should use the schemes as follows when they want to establish keying data. First, entities U and V should use the setup procedure to establish which options to use the scheme with, then they should use the key deployment procedure to select key pairs and exchange public keys. Finally, when U and V want to establish keying data they should simultaneously use the key agreement operation. Provided U and V operate the key agreement operation with corresponding keys as inputs, they will obtain the same keying data.

Note that this document does not address how specific keys should be derived from keying data established using a key agreement scheme. This detail is left to be determined on an application by application basis. Some applications may wish simply to use the keying data directly as a key, others may wish to split the keying data into more than one key, and others may wish to process the keying data to exclude weak keys.

Key agreement schemes are designed to meet a wide variety of security goals depending on how they are applied — security goals that the schemes described here are designed to provide include unilateral implicit key authentication, mutual implicit key authentication, known-key security, and forward secrecy, in the presence of adversaries capable of launching both passive and active attacks.

Two key agreement schemes are supported at this time: the elliptic curve Diffie-Hellman scheme, and the elliptic curve MQV scheme. The elliptic curve Diffie-Hellman scheme is specified in Section 6.1, and the elliptic curve MQV scheme is specified in Section 6.2.

See Appendix B for a commentary on the contents on this section, including implementation discussion, security discussion, and references.

6.1 Elliptic Curve Diffie-Hellman Scheme

The elliptic curve Diffie-Hellman scheme is a key agreement scheme based on ECC. It is designed to provide a variety of security goals depending on its application — goals it can provide include unilateral implicit key authentication, mutual implicit key authentication, known-key security, and forward secrecy — depending on factors such as whether or not public keys are exchanged in an authentic manner, and whether key pairs are ephemeral or static. See Appendix B for a further discussion.

The setup procedure for the elliptic curve Diffie-Hellman scheme is specified in Section 6.1.1, the key deployment procedure is specified in Section 6.1.2, and the key agreement operation is specified in Section 6.1.3.

6.1.1 Scheme Setup

Entities U and V should perform the following setup procedure to prepare to use the elliptic curve Diffie-Hellman scheme:

1. Entities U and V should establish which of the key derivation functions supported in Section 3.6 to use, and select any options involved in the operation of the key derivation function. Let KDF denote the key derivation function chosen.

2. Entities U and V should establish whether to use the "standard" elliptic curve Diffie-Hellman primitive specified in Section 3.3.1, or the elliptic curve cofactor Diffie-Hellman primitive specified in Section 3.3.2.

3. Entities U and V should establish at the desired security level elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$. The elliptic curve domain parameters T should be generated using the primitive specified in Section 3.1.1.1 or the primitive specified in Section 3.1.2.1. Both U and V should receive an assurance that the elliptic curve domain parameters T are valid using one of the methods specified in Section 3.1.1.2 or Section 3.1.2.2.

6.1.2 Key Deployment

Entities U and V should perform the following key deployment procedure to prepare to use the elliptic curve Diffie-Hellman scheme:

1. Entity U should establish an elliptic curve key pair (d_U, Q_U) associated with the elliptic curve domain parameters T established during the setup procedure. The key pair should be generated using the primitive specified in Section 3.2.1.

2. Entity V should establish an elliptic curve key pair (d_V, Q_V) associated with the elliptic curve domain parameters T established during the setup procedure. The key pair should be generated using the primitive specified in Section 3.2.1.

3. Entities U and V should exchange their public keys Q_U and Q_V.

4. If the "standard" elliptic curve Diffie-Hellman primitive is being used, U should receive an assurance that Q_V is valid using one of the methods specified in Section 3.2.2, and if the elliptic curve cofactor Diffie-Hellman primitive is being used, U should receive an assurance that Q_V is at least partially valid using one of the methods specified in Section 3.2.2 or Section 3.2.3.

5. If the "standard" elliptic curve Diffie-Hellman primitive is being used, V should receive an assurance that Q_U is valid using one of the methods specified in Section 3.2.2, and if the elliptic curve cofactor Diffie-Hellman primitive is being used, V should receive an assurance that Q_U is at least partially valid using one of the methods specified in Section 3.2.2 or Section 3.2.3.

6.1.3 Key Agreement Operation

Entities U and V should perform the key agreement operation described in this section to establish keying data using the elliptic curve Diffie-Hellman scheme. For clarity, only U's use of the operation is described. Entity V's use of the operation is analogous, but with the roles of U and V reversed.

Entity U should establish keying data with V using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: The input to the key agreement operation is:

1. An integer *keydatalen* which is the number of octets of keying data required.
2. (Optional) An octet string *SharedInfo* which consists of some data shared by U and V.

Output: An octet string K which is the keying data of length *keydatalen* octets, or "invalid".

Actions: Establish keying data as follows:

1. Use one of the Diffie-Hellman primitives specified in Section 3.3 to derive a shared secret field element $z \in \mathbb{F}_q$ from U's secret key d_U established during the key deployment procedure and V's public key Q_V obtained during the key deployment procedure. If the Diffie-Hellman primitive outputs "invalid", output "invalid" and stop. Decide whether to use the "standard" elliptic curve Diffie-Hellman primitive or the elliptic curve cofactor Diffie-Hellman primitive according to the convention established during the setup procedure.
2. Convert $z \in \mathbb{F}_q$ to an octet string Z using the conversion routine specified in Section 2.3.5.
3. Use the key derivation function *KDF* established during the setup procedure to generate keying data K of length *keydatalen* octets from Z and [*SharedInfo*]. If the key derivation function outputs "invalid", output "invalid" and stop.
4. Output K.

6.2 Elliptic Curve MQV Scheme

The elliptic curve MQV scheme is a key agreement scheme based on ECC. It is designed to provide a variety of security goals depending on its application — goals it can provide include mutual implicit key authentication, known-key security, and forward secrecy — depending on factors such as whether or not U and V both contribute ephemeral key pairs. See Appendix B for a further discussion.

The setup procedure for the elliptic curve MQV scheme is specified in Section 6.2.1, the key deployment procedure is specified in Section 6.2.2, and the key agreement operation is specified in Section 6.2.3.

6.2.1 Scheme Setup

Entities U and V should perform the following setup procedure to prepare to use the elliptic curve MQV scheme:

1. Entities U and V should establish which of the key derivation functions supported in Section 3.6 to use, and select any options involved in the operation of the key derivation function. Let KDF denote the key derivation function chosen.

2. Entities U and V should establish at the desired security level elliptic curve domain parameters $T = (p, a, b, G, n, h)$ or $(m, f(x), a, b, G, n, h)$. The elliptic curve domain parameters T should be generated using the primitive specified in Section 3.1.1.1 or the primitive specified in Section 3.1.2.1. Both U and V should receive an assurance that the elliptic curve domain parameters T are valid using one of the methods specified in Section 3.1.1.2 or Section 3.1.2.2.

6.2.2 Key Deployment

Entities U and V should perform the following key deployment procedure to prepare to use the elliptic curve MQV scheme:

1. Entity U should establish two elliptic curve key pairs $(d_{1,U}, Q_{1,U})$ and $(d_{2,U}, Q_{2,U})$ associated with the elliptic curve domain parameters T established during the setup procedure. The key pairs should both be generated using the primitive specified in Section 3.2.1.

2. Entity V should establish two elliptic curve key pairs $(d_{1,V}, Q_{1,V})$ and $(d_{2,V}, Q_{2,V})$ associated with the elliptic curve domain parameters T established during the setup procedure. The key pairs should both be generated using the primitive specified in Section 3.2.1.

3. Entity U should obtain in an authentic manner the first elliptic curve public key $Q_{1,V}$ selected by V. Entity U should receive an assurance that $Q_{1,V}$ is valid using one of the methods specified in Section 3.2.2.

4. Entity V should obtain in an authentic manner the first elliptic curve public key $Q_{1,U}$ selected by U. Entity V should receive an assurance that $Q_{1,U}$ is valid using one of the methods specified in Section 3.2.2.

5. Entities U and V should exchange their second public keys $Q_{2,U}$ and $Q_{2,V}$.

6. Entity U should receive an assurance that $Q_{2,V}$ is at least partially valid using one of the methods specified in Section 3.2.2 or Section 3.2.3.

7. Entity V should receive an assurance that $Q_{2,U}$ is at least partially valid using one of the methods specified in Section 3.2.2 or Section 3.2.3.

6.2.3 Key Agreement Operation

Entities U and V should perform the key agreement operation described in this section to establish keying data using the elliptic curve MQV scheme. For clarity, only U's use of the operation is described. Entity V's use of the operation is analogous, but with the roles of U and V reversed.

Entity U should establish keying data with V using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: The input to the key agreement operation is:

1. An integer *keydatalen* which is the number of octets of keying data required.
2. (Optional) An octet string *SharedInfo* which consists of some data shared by U and V. This octet string should be included, and should contain information identifying the entities U and V.

Output: An octet string K which is the keying data of length *keydatalen* octets, or "invalid".

Actions: Establish keying data as follows:

1. Use the elliptic curve MQV primitive specified in Section 3.4 to derive a shared secret field element $z \in \mathbb{F}_q$ from U's key pairs $(d_{1,U}, Q_{1,U})$ and $(d_{2,U}, Q_{2,U})$ established during the key deployment procedure and V's public keys $Q_{1,V}$ and $Q_{2,V}$ obtained during the key deployment procedure. If the MQV primitive outputs "invalid", output "invalid" and stop.
2. Convert $z \in \mathbb{F}_q$ to an octet string Z using the conversion routine specified in Section 2.3.5.
3. Use the key derivation function *KDF* established during the setup procedure to generate keying data K of length *keydatalen* octets from Z and [*SharedInfo*]. If the key derivation function outputs "invalid", output "invalid" and stop.
4. Output K.

A Glossary

This section supplies a glossary to the terms and notation used in this document.

The section is organized as follows. Section A.1 lists the terms used in this document, Section A.2 lists the acronyms used, and Section A.3 specifies the notation used.

A.1 Terms

Terms used in this document include:

active attack	The ability of an adversary of a cryptographic scheme to subvert communications between entities by deleting, injecting, substituting, or generally subverting messages in any way.
addition rule	An addition rule describes the addition of two elliptic curve points P_1 and P_2 to produce a third elliptic curve point P_3. See Section 2.2.
base point	A distinguished point G on an elliptic curve.
binary polynomial	A polynomial whose coefficients are either 0's or 1's.
bit string	A bit string is an ordered sequence of 0's and 1's.
certificate	The public key and identity of an entity together with some other information, rendered unforgeable by signing the certificate with the secret key of a Certification Authority.
Certification Authority	A Center trusted by one or more entities to create and assign certificates.
characteristic 2 finite field	A finite field containing 2^m elements, where $m \geq 1$ is an integer.
chosen-ciphertext attack	The ability of an adversary of an encryption scheme to obtain the decryptions of ciphertexts of its choice in an attempt to compromise the scheme.
chosen-message attack	The ability of an adversary of a signature scheme to obtain signatures on messages of its choice in an attempt to compromise the scheme. Similarly the ability of an adversary of a MAC scheme to obtain tags on messages of its choice in an attempt to compromise the scheme.
chosen-plaintext attack	The ability of an adversary of an encryption scheme to obtain the encryptions of plaintexts of its choice in an attempt to compromise the scheme.
ciphertext	The result of applying an encryption operation to a message.

cryptographic scheme	A cryptographic scheme consists of an unambiguous specification of a set of operations capable of providing a security service when properly implemented and maintained.
data confidentiality	The assurance that data is unintelligible to unauthorized parties.
data integrity	The assurance that data has not been modified by unauthorized parties.
data origin authentication	The assurance that the purported origin of data is correct.
elliptic curve	An elliptic curve over $\mathbb{F}_q$ is a set of points which satisfy a certain equation specified by two parameters a and b, which are elements of the field $\mathbb{F}_q$. See Section 2.2.
elliptic curve domain parameters	Elliptic curve domain parameters are comprised of a field size q, a reduction polynomial $f(x)$ in the case $q = 2^m$, two elements a, b in $\mathbb{F}_q$ which define an elliptic curve E over $\mathbb{F}_q$, a point G of prime order in $E(\mathbb{F}_q)$, the order n of G, and the cofactor h. See Section 3.1.
elliptic curve key pair	Given particular elliptic curve domain parameters, an elliptic curve key pair (d, Q) consists of an elliptic curve secret key d and the corresponding elliptic curve public key Q.
elliptic curve point	If E is an elliptic curve defined over $\mathbb{F}_q$, then an elliptic curve point P is either a pair of field elements (x_P, y_P) (where $x_P, y_P \in \mathbb{F}_q$) such that the values $x = x_P$ and $y = y_P$ satisfy the equation defining E, or a special point $\mathcal{O}$ called the point at infinity.
elliptic curve public key	Given particular elliptic curve domain parameters, and an elliptic curve secret key d, the corresponding elliptic curve public key Q is the elliptic curve point $Q = dG$, where G is the base point.
elliptic curve secret key	Given particular elliptic curve domain parameters, an elliptic curve secret key d is an integer in the interval $[1, n-1]$, where n is the prime order of the base point G.
encryption scheme	An encryption scheme is a cryptographic scheme consisting of an encryption operation and a decryption operation which is capable of providing data confidentiality.
entity	A party involved in the operation of a cryptographic system.
ephemeral	Ephemeral data is relatively short-lived. Usually ephemeral data refers to data specific to a particular execution of a cryptographic scheme.

existentially unforgeable	A signature scheme is existentially unforgeable if it is infeasible for an adversary to forge a signature on any message that has not previously been signed by the scheme's legitimate user. Similarly a MAC scheme is existentially unforgeable if it is infeasible for an adversary to forge the tag on any message that has not previously been tagged by one of the scheme's legitimate users.
forward secrecy	The assurance that a session key established between some parties will not be compromised in the event that some of the parties' static secret keys are compromised in the future. Also known as perfect forward secrecy.
hash function (cryptographic hash function)	A cryptographic hash function is a function which maps bit strings from a large (possibly very large) domain into a smaller range. The function possesses the following properties: it is computationally infeasible to find any input which maps to any pre-specified output, and it is computationally infeasible to find any two distinct inputs which map to the same output.
implicit key authentication	A key establishment scheme provides implicit key authentication if only authorized parties are possibly capable of computing any session key.
irreducible binary polynomial	A binary polynomial $f(x)$ is irreducible if it does not factor over $\mathbb{F}_2$ as a product of two or more binary polynomials, each of degree less than the degree of $f(x)$.
key (cryptographic key)	A parameter that determines the execution of a cryptographic operation such as the transformation from plaintext to ciphertext and vice versa, the synchronized generation of keying material, or a digital signature computation or validation.
key agreement scheme	A key agreement scheme is a key establishment scheme in which the keying data established is a function of contributions provided by each party to the scheme in such a way that no party can predetermine the value of the keying data.
key derivation function	A key derivation function is a function which takes as input a shared secret value and outputs keying data suitable for later cryptographic use.
key establishment scheme	A key establishment scheme is a cryptographic scheme which reveals to its legitimate users keying data suitable for subsequent use in cryptographic schemes.
keying data	Data suitable for use as cryptographic keys.
known-key security	The assurance that a session key established by an execution of a key establishment scheme will not be compromised in the event that other session keys are compromised.

MAC scheme	A MAC scheme is a cryptographic scheme consisting of a message tagging operation and a tag checking operation which is capable of providing data origin authentication and data integrity.
non-repudiation	The assurance that the origin and integrity of data can be proved to a third party.
octet	An octet is a bit string of length 8. An octet is represented by a hexadecimal string of length 2. The first hexadecimal digit represents the four leftmost bits of the octet, and the second hexadecimal digit represents the four rightmost bits of the octet. For example, $9D$ represents the bit string 10011101. An octet also represents an integer in the interval $[0, 255]$. For example, $9D$ represents the integer 157.
octet string	An octet string is an ordered sequence of octets.
order of a curve	The order of an elliptic curve E defined over the field $\mathbb{F}_q$ is the number of points on the elliptic curve E, including $\mathcal{O}$. This is denoted by $\#E(\mathbb{F}_q)$.
order of a point	The order of a point P is the smallest positive integer n such that $nP = \mathcal{O}$ (the point at infinity).
partially valid elliptic curve public key	An elliptic curve public key $Q = (x_Q, y_Q)$ is partially valid if the values $x = x_Q$ and $y = y_Q$ satisfy the defining equation of the associated elliptic curve E, but it is not necessarily the case that $Q = dG$ for some d. The elliptic curve public key partial validation primitive in Section 3.2.3.1 checks whether or not an elliptic curve public key is partially valid.
passive attack	The ability of an adversary of a cryptographic scheme merely to observe the communications of entities using the scheme.
pentanomial	A binary polynomial of the form $x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$, where $1 \leq k_1 < k_2 < k_3 \leq m - 1$.
plaintext	A message to be encrypted; the opposite of ciphertext.
plaintext-awareness	An encryption scheme is plaintext-aware if it is infeasible to generate a valid ciphertext without knowing the corresponding message.
point compression	Point compression allows a point $P = (x_P, y_P)$ to be represented compactly using x_P and a single additional bit $\tilde{y}_P$ derived from x_P and y_P. See Section 2.3.
prime finite field	A finite field containing p elements, where p is an odd prime number.
primitive (cryptographic primitive)	A cryptographic primitive is an operation not by itself capable of providing a security service, but which can be incorporated in a cryptographic scheme.

public key	In a public-key scheme, that key of an entity's key pair which can be publicly known.
public-key cryptographic scheme	A cryptographic scheme in which each operation is controlled by one of two keys; either the public key or the private key. The two keys have the property that, given the public key, it is computationally infeasible to derive the private key. Also known as asymmetric cryptographic scheme.
reduction polynomial	The irreducible binary polynomial $f(x)$ of degree m that is used to determine a representation of $\mathbb{F}_{2^m}$.
scalar multiplication	If k is a positive integer, then $k \times P$ or kP denotes the point obtained by adding together k copies of the point P. The process of computing kP from P and k is called scalar multiplication.
secret key	In a public-key system, that key of an entity's key pair which must be known only by that entity. Also known as private key.
semantically secure	An encryption scheme is semantically secure if it is infeasible for an adversary to learn anything from ciphertext about the corresponding plaintext apart from the length of the plaintext.
session key	A key (usually short-lived) established using a key establishment scheme.
shared secret value	An intermediate value in a key establishment scheme from which keying data is derived.
signature scheme	A signature scheme is a cryptographic scheme consisting of a signing operation and a verifying operation and which is capable of providing data origin authentication, data integrity, and non-repudiation.
static	Static data is relatively long-lived. Usually static data refers to data common to a number of executions of a cryptographic scheme.
symmetric cryptographic scheme	A cryptographic scheme in which each operation is controlled by the same key.
trinomial	A binary polynomial of the form $x^m + x^k + 1$, where $1 \leq k \leq m - 1$.
unknown key-share resilience	The assurance that all the parties who share a session key are aware of the identities of the parties with which they share the key.

valid elliptic curve domain parameters	Elliptic curve domain parameters are valid if they satisfy the arithmetic requirements of elliptic curve domain parameters. Equivalently, elliptic curve domain parameters are valid if they have been generated as specified in Section 3.1.1.1 or 3.1.2.1. The elliptic curve domain parameter validation primitives in Sections 3.1.1.2.1 and 3.1.2.2.1 check whether or not elliptic curve domain parameters are valid.
valid elliptic curve public key	An elliptic curve public key $Q = (x_Q, y_Q)$ is valid if it satisfies the arithmetic requirements of an elliptic curve public key — namely that $Q = dG$ for some d in the interval $[1, n-1]$ where G is the base point of the associated elliptic curve domain parameters and n is the order of G. The elliptic curve public key validation primitive in Section 3.2.2.1 checks whether or not an elliptic curve public key is valid.
x-coordinate	The x-coordinate of an elliptic curve point, $P = (x_P, y_P)$, is x_P.
y-coordinate	The y-coordinate of an elliptic curve point, $P = (x_P, y_P)$, is y_P.

A.2 Acronyms, Initialisms and Other Abbreviations

The acronyms, initialisms and other abbreviations used in this document denote:

AES	Advanced Encryption Standard. See [197].
ANS	American National Standard.
ANSI	American National Standards Institute.
ASC X9	Accredited Standards Committee X9.
ASN.1	Abstract Syntax Notation One.
CA	Certification Authority. See [3279].
CBC	Cipher Block Chaining.
CMAC	Cipher-based Message Authentication Code
CMS	Cryptographic Message Syntax. See [2630].
CTR	Counter (block cipher mode of operation)
CRL	Certificate Revocation List. See [3279].
DER	Distinguished Encoding Rules. See [X.690].
DES	Data Encryption Standard. See [46-2].
DSA	Digital Signature Algorithm. See [186-2].
DSS	Digital Signature Standard. See [186-2].
EC	Elliptic Curve.

ECC	Elliptic Curve Cryptography.
ECIES	Elliptic Curve Integrated Encryption Scheme. See Section 5.1.
ECDH	Elliptic Curve Diffie-Hellman. See Section 6.1.
ECDHP	Elliptic Curve Diffie-Hellman Problem.
ECDLP	Elliptic Curve Discrete Logarithm Problem.
ECDSA	Elliptic Curve Digital Signature Algorithm. See Section 4.1.
ECMQV	Elliptic Curve Menezes-Qu-Vanstone. See Section 6.2.
ECWKTS	Elliptic Curve Wrapped Key Transport Scheme. See Section 5.2
FIPS	Federal Information Processing Standard.
GEC	Guideline for Efficient Cryptography.
HMAC	Hash-based Message Authentication Code. See [2104].
IACR	International Association for Cryptologic Research
IEC	International Electrotechnical Commission.
IEEE	Institute of Electrical and Electronics Engineers.
IETF	Internet Engineering Task Force.
IKE	Internet Key Exchange.
ISO	International Organization for Standardization.
ITU	International Telecommunications Union.
LNCS	Lecture Notes in Computer Science
KDF	Key Derivation Function.
MQV	Menezes-Qu-Vanstone. See [LMQ$^+$98].
NIST	(U.S.) National Institute of Standards and Technology.
PKI	Public Key Infrastructure.
PKIX	Public Key Infrastructure for the Internet.
RFC	Request for Comment.
RNG	Random Number Generator.
SEC	Standard for Efficient Cryptography
SHA-1	Secure Hash Algorithm Revision One. See [180-1].
SSL	Secure Sockets Layer.
TDES	Triple Data Encryption Standard. See [X9.52].
TLS	Transport Layer Security.
WAP	Wireless Application Protocol.
WTLS	Wireless Transport Layer Security.
XOR	Exclusive Or

A.3 Notation

The notation adopted in this document is:

$[X]$	Indicates that the inclusion of X is optional.
$[x, y]$	The interval of integers between and including x and y.
$\lceil x \rceil$	Ceiling: the smallest integer $\geq x$. For example, $\lceil 5 \rceil = 5$ and $\lceil 5.3 \rceil = 6$.
$\lfloor x \rfloor$	Floor: the largest integer $\leq x$. For example, $\lfloor 5 \rfloor = 5$ and $\lfloor 5.3 \rfloor = 6$.
$x \bmod n$	The unique remainder r, $0 \leq r \leq n-1$, when x is divided by n. For example, $23 \bmod 7 = 2$.
C	Ciphertext.
d	An EC private key.
E	An elliptic curve over the field $\mathbb{F}_q$ defined by a and b.
$E(\mathbb{F}_q)$	The set of all points on an elliptic curve E defined over $\mathbb{F}_q$ and including the point at infinity $\mathcal{O}$.
$\#E(\mathbb{F}_q)$	If E is defined over $\mathbb{F}_q$, then $\#E(\mathbb{F}_q)$ denotes the number of points on the curve (including the point at infinity $\mathcal{O}$). $\#E(\mathbb{F}_q)$ is called the order of the curve E.
$\mathbb{F}_{2^m}$	The finite field containing 2^m elements, where m is a positive integer.
$\mathbb{F}_p$	The finite field containing p elements, where p is a prime.
$\mathbb{F}_q$	The finite field containing q elements. In this document attention is restricted to the cases where q is an odd prime number (p) or a power of 2 (2^m).
G	A distinguished point on an elliptic curve called the base point or generating point.
$\gcd(x, y)$	The greatest common divisor of integers x and y.
h	$h = \#E(\mathbb{F}_q)/n$, where n is the order of the base point G. h is called the co-factor.
k	An EC private key specific to one particular instance of a cryptographic scheme.
K	Keying data.
$\log_2 x$	The logarithm of x to the base 2.
m	The degree of the finite field $\mathbb{F}_{2^m}$.
M	A message.
mod	Modulo.

$\mod f(x)$	Arithmetic modulo the polynomial $f(x)$.
$\mod n$	Arithmetic modulo n.
n	The order of the base point G.
$\mathcal{O}$	A special point on an elliptic curve, called the point at infinity. This is the additive identity of the elliptic curve group.
p	An odd prime number.
P	An EC point.
q	The number of elements in the field $\mathbb{F}_q$.
Q	An EC public key.
R	An EC public key specific to one particular instance of a cryptographic scheme.
S	A digital signature.
T	Elliptic curve domain parameters.
U, V	Entities.
$\|X\|$	Length in octets of the octet string X.
$X \| Y$	Concatenation of two strings X and Y. X and Y are either both bit strings or both octet strings.
$X \oplus Y$	Bitwise exclusive-or of two bit strings X and Y of the same bit length.
x_P	The x-coordinate of a point P.
y_P	The y-coordinate of a point P.
$\tilde{y}_P$	The representation of the y-coordinate of a point P when point compression is used.
z, or Z	A shared secret value. z denotes a shared secret integer or field element, and Z a shared secret bit string or octet string.

Furthermore positional notation is used to indicate the association of a value to a particular entity. For example d_U denotes an EC private key owned by entity U. Occasionally positional notation is also used to indicate a counter value associated with some data, or to indicate the base in which a particular value is being expressed if there is some possibility of ambiguity. For example, $Hash_1$ denotes the value of $Hash_i$ when the counter i has value 1, and 01_{16} denotes that the value 01 is written in hexadecimal.

With the exception of notation that has been well-established in other documents, where possible in this document capital letters are used in variable names that denote bit strings or octet strings, and capital letters are excluded from variable names that denote field elements or integers. For example, d is used to denote the integer that specifies an EC private key, and M is used to denote the octet string to be signed using a signature scheme.

B Commentary

This section provides commentary on the main body of this document, including implementation discussion, security discussion, and references.

The aim of this section is to supply implementers with relevant guidance. However the section does not attempt to provide exhaustive information but rather focuses on giving basic information and including pointers to references which contain additional material. Furthermore the section concentrates on supplying information specific to ECC rather than providing extensive information on components such as SHA-1 and TDES which are specified elsewhere.

The information in this section is current as of November 2008. The information is likely to change over time, and implementers should therefore survey the state-of-the-art at the time of implementation and carry out periodic reviews subsequent to deployment.

Excellent treatments focusing on ECC are contained in Blake, Seroussi, and Smart [BSS99, BSS05], Hankerson, Menezes, and Vanstone [HMV04], Koblitz [Kob94], Cohen and Frey *et al.* [CFA+06], and Menezes [Men93].

This section is organized as follows. Sections B.1 through B.5 respectively provide commentary on Sections 2 through 6 of the main body of this document. Section B.6 supplies information regarding the alignment of this document with other standards efforts which include ECC.

B.1 Commentary on Section 2 — Mathematical Foundations

This section provides commentary on Section 2 of the main body of this document.

Finite fields and elliptic curves have been studied as mathematical objects for hundreds of years. The body of literature on these structures is vast. Introductions to finite fields can be found in the books of Jungnickel [Jun93], Lidl and Niederreiter [LN87], and McEliece [McE87]. An introduction to elliptic curves can be found in the book of Silverman [Sil85].

Elliptic curves over finite fields were first proposed for use to build cryptographic schemes in 1985 by Koblitz [Kob87] and Miller [Mil85].

The security of all cryptographic schemes based on ECC relies on the elliptic curve discrete logarithm problem or ECDLP. The ECDLP is stated as follows in the case of interest here — namely when the elliptic curve in question has order divisible by a large prime n.

Let E be an elliptic curve defined over a finite field $\mathbb{F}_q$, and let $G \in E(\mathbb{F}_q)$ be a point on E of large prime order n. The ECDLP is, given E, G, and a scalar multiple Q of G, to determine an integer l such that $Q = lG$.

No general subexponential algorithms are known for the ECDLP. The best general algorithms known to date are based on the Pollard-ρ method and the Pollard-λ method [Pol78]. The Pollard-ρ method takes about $\sqrt{\pi n/2}$ steps, and the Pollard-λ method takes about $2\sqrt{n}$ steps. A *step* here is roughly a single elliptic curve group operation. Both methods can be parallelized effectively — see [vOW94].

Gallant, Lambert, and Vanstone [GLV00], and Wiener and Zuccherato [WZ99] showed that the

Pollard-ρ method can be sped up by a factor of $\sqrt{2}$. Thus the expected running time of the Pollard-ρ method with this speedup is $\sqrt{\pi n/4}$ steps.

They also showed that this speedup can be enhanced when E is an elliptic curve over $\mathbb{F}_{2^{ed}}$ which is defined over $\mathbb{F}_{2^e}$. In this case they show that the Pollard-ρ method can be sped up by a factor of $\sqrt{2d}$. For example, the Koblitz curve $E: y^2 + xy = x^3 + x^2 + 1$ over $\mathbb{F}_{2^{163}}$ has the property that $\#E(\mathbb{F}_{2^{163}}) = 2n$ where n is a 162-bit prime. As a result of the enhancement to the Pollard-ρ method, the ECDLP in $E(\mathbb{F}_{2^{163}})$ can be solved in about 2^{77} steps as opposed to the 2^{81} steps required to solve the ECDLP for a random curve of similar order.

Table 2 below illustrates the difficulty of the ECDLP. It contains estimates in MIPS years of the computing power required to solve the ECDLP on a general curve in software using the improved Pollard-ρ method. To place Table 2 in context, Odlyzko has estimated that 0.1% of the world's computing power working for 1 year will amount to 10^8 MIPS years in 2004, and 10^{10} or 10^{11} MIPS years in 2014 [Odl95]. Table 2 is reproduced from ANS X9.62 [X9.62b]. More details on how the estimates were obtained can be found there.

Size of n (in bits)	$\sqrt{\pi n/4}$	MIPS years
160	2^{80}	8.5×10^{11}
192	2^{96}	5.6×10^{16}
224	2^{112}	3.7×10^{21}
256	2^{128}	2.4×10^{26}
384	2^{192}	4.4×10^{45}
521	2^{260}	1.3×10^{66}

Table 2: Computing power required to solve ECDLP

The difficulty of the ECDLP is further illustrated by van Oorschot and Wiener's 1994 paper [vOW94]. Van Oorschot and Wiener carried out a detailed study of the feasibility of building special-purpose hardware to solve the ECDLP. They estimated that for about $10 million a machine with 325,000 processors could be built that would solve the ECDLP for an elliptic curve E with $n \approx 2^{120}$ in about 35 days. Hardware attacks on larger values of n, such as $n \approx 2^{160}$, appear impractical at this time. Pelzl [Pel06] estimated in 2006 that the cost of special purpose hardware to solve the ECDLP for $n \approx 2^{160}$ over a prime field is about $\$6 \times 10^{11}$.

Finally, although no general subexponential algorithms to solve the ECDLP are known, three classes of curves are susceptible to special-purpose algorithms. Firstly, elliptic curves E over $\mathbb{F}_q$ with n dividing $q^B - 1$ for small B are susceptible to the attacks described by Menezes, Okamoto, and Vanstone [MOV93], and Frey and Rück [FR94]. The attacks efficiently reduce the ECDLP on these curves to the traditional discrete logarithm problem in a small degree extension of $\mathbb{F}_q$. A bound $B \geq 20$ was updated to $B \geq 100$ in [X9.62a] to provide a large margin for safety. Galbraith [Gal05], Koblitz and Menezes [KM05], and Hitt [Hit07] note further that if $q = p^m$, then one may also need to consider fractional B, with denominator dividing m. Secondly, elliptic curves E over $\mathbb{F}_q$ with $\#E(\mathbb{F}_q) = q$ are susceptible to the attack described by Semaev [Sem98], Smart [Sma99],

and Satoh and Araki [SA98]. This attack efficiently maps the elliptic curve group $E(\mathbb{F}_q)$ into the additive group of $\mathbb{F}_q$. Thirdly, for curves defined over $\mathbb{F}_q$ where $q = 2^m$ with m composite, certain attacks based on Weil descent, and more recently, index calculus, have been discovered. This is an ongoing research area. On a precautionary basis, however, such curves should be avoided. All known weak classes of curves are excluded in this document.

Cheon [Che06], expanding on the work of Brown and Gallant [BG04a], describes an exponential time algorithm that, for certain elliptic curves, use about $\sqrt[3]{n}$ computation and $\sqrt[3]{n}$ invocations of a user's Diffie-Hellman primitive to find the user's private key. The conditions under which this attack is possible are that one of $n-1$ or $n+1$ has a factor of size approximately $\sqrt[3]{n}$. One way to avoid this attack is to not provide an adversary $\sqrt[3]{n}$ accesses to the Diffie-Hellman primitive. Indeed, in Diffie-Hellman key agreement and ECIES, the adversary gets no access to the Diffie-Hellman primitive of the user, because a one-way key derivation function is applied. Of course, the number of accesses to the primitive $\sqrt[3]{n}$ is extremely large, so in practice is unlikely to be feasible for an adversary. Nevertheless, as a precautionary measure, one may want to choose elliptic curve domain parameters that resist Cheon's attack by arranging that $n-1$ and $n+1$ have very large prime factors. Although it may seem optimal to choose cofactors 2 and 4, Brown and Gallant argue that a slightly larger cofactor may provide the following theoretical benefit. Given a cofactor of a certain size, it can be shown that breaking the elliptic curve Diffie-Hellman primitive is roughly as difficult as solving the discrete logarithm problem.

Additional information on the difficulty of the ECDLP can be found in ANS X9.62 [X9.62b], ANS X9.63 [X9.63], Blake, Seroussi, and Smart [BSS99, BSS05], Hankerson, Menezes, and Vanstone [HMV04], Galbraith and Menezes [GM05], and Cohen and Frey *et al.* [CFA+06]. A useful source of information on the state-of-the-art in practical attacks on the ECDLP is Certicom's ECC challenge [ECC99].

The efficiency of cryptographic schemes based on ECC usually rests primarily on the efficiency of field operation computations and in particular scalar multiplication computation. Efficient general techniques for computing field operations are well-known and are described in, for example, [HMV04, Knu81, McE87, MvOV97]. A variety of efficient general techniques for computing scalar multiplication are known such as switching to projective coordinates and using pre-computation.

Both field operation computations and scalar multiplication computation can be accelerated by choosing particular underlying fields and elliptic curves. Examples of fields amenable to particularly efficient implementation include $\mathbb{F}_{2^m}$ and $\mathbb{F}_p$ where p is a Mersenne or generalized Mersenne prime — see, for example [ABMV93, AMV93, AMOV91, Nat99]. (Solinas [Sol01] describes some special curves with cofactor larger than four, which is why a cofactor larger than four has been allowed in this version of the standard.) Examples of elliptic curves amenable to particularly efficient implementation include Koblitz curves over $\mathbb{F}_{2^m}$ [Kob91] which possess an efficiently computable endomorphism.

Additional information on the implementation of efficient finite field operations and scalar multiplication can be found in Blake, Seroussi, and Smart [BSS99, BSS05], Hankerson, Menezes, and Vanstone [HMV04], and Cohen and Frey *et al.* [CFA+06].

B.2 Commentary on Section 3 — Cryptographic Components

This section provides commentary on Section 3 of the main body of this document.

B.2.1 Commentary on Elliptic Curve Domain Parameters

Elliptic curve domain parameters must be generated during the setup procedure of each of the schemes specified in this document.

The first step in this process is to determine the security level desired by the application in question. A number of criteria may affect this determination — including, for example, the value of the information that the scheme will be used to protect, the length of time the parameters will be used for, and the security level of other schemes used in the application.

Table 3 below provides additional information which may help determine the security level desired. It lists comparable key sizes for an "ideal" symmetric scheme, an ECC-based scheme, and a scheme such as DSA or RSA based on the integer factorization problem or traditional discrete logarithm problem.

Security level	Symmetric	ECC	DSA/RSA	Protects to year
80	80	160	1024	2010
112	112	224	2048	2030
128	128	256	3072	2040
192	192	384	7680	2080
256	256	512	15360	2120

Table 3: Comparable key sizes

Once the desired security level has been selected, there are a number of ways to generate elliptic curve domain parameters at a given strength. These include:

- Select an appropriate finite field. Then select an elliptic curve over the field at random. Count the number of points on the curve using Schoof's algorithm [Sch85], or one of its various improvements, such as the Schoof-Elkies-Atkin (SEA) algorithm [Elk98, Atk92] and Satoh's algorithm [Sat00] and its improvements discussed in [Ver05]. Check whether the number of points is nearly prime. Repeat until appropriate parameters are found.
- Select an appropriate field. Then select an appropriate curve order, and generate a curve over the field with this number of points using techniques based on "complex multiplication" [LZ94].
- Select an appropriate finite field. Then select an elliptic curve over the field from a special family of curves whose order is easily computable (such as the family of Koblitz curves). Compute the number of points on this curve, and check whether the number of points is nearly prime. Repeat until appropriate parameters are found.

The first method — known as selecting elliptic curve domain parameters at random — is the most conservative choice because it offers a probabilistic guarantee against future special purpose attacks of a similar nature to the Menezes-Okamoto-Vanstone and the Semaev-Smart-Satoh-Araki attacks described in Section B.1. However, existing implementations of Schoof's algorithm are less efficient than implementations of the other parameter selection methods (but in the case of Satoh's algorithm and its improvements, the discrepancy may be negligible). The second method — known as selecting elliptic curve domain parameters using complex multiplication — generates parameters more efficiently than the first method. The third method — known as selecting elliptic curve domain parameters from a special family — also generates parameters more efficiently than the first method, and has the added attraction than some special families of curves (such as the family of Koblitz curves) enable acceleration of computations such as scalar multiplication. However despite their efficiency benefits, the second and third methods should be used with a good deal of caution because they produce parameters which may be susceptible to future special-purpose attacks.

An attractive refinement of the idea of selecting elliptic curve domain parameters at random is the idea of selecting elliptic curve domain parameters verifiably at random. This involves selecting parameters at random from a seed in such a way that the parameters cannot be pre-determined. It is appealing because the seed provides evidence that can be verified by anyone that no "trapdoors" have been placed in the parameters. One method of selecting parameters verifiably at random is specified in Section 3.1.3 and also in ANS X9.62 .

SEC 2 [SEC 2] — a companion document to this document — provides a list of precomputed elliptic curve domain parameters at a variety of commonly required security levels that implementers may use when implementing the schemes in this document. Use of these precomputed parameters is strongly recommended in order to foster interoperability.

Once elliptic curve domain parameters have been generated, either by the users themselves or by a third party, it is desirable to receive some assurance that the parameters are valid: that the parameters possess the arithmetic properties expected from secure parameters. Parameter validation mitigates against inadvertent generation of insecure parameters caused by coding errors, and against deliberate attempts to trick users into using insecure parameters.

Additional information on the generation and validation of elliptic curve domain parameters can be found in ANS X9.62 [X9.62b], ANS X9.63 [X9.63], and IEEE 1363 [1363].

B.2.2 Commentary on Elliptic Curve Key Pairs

Elliptic curve key pairs must be generated during the operation of each of the schemes specified in this document. The key pair generation process requires a secure random or pseudorandom number generator. Design of secure random and pseudorandom number generators is difficult, so implementers should therefore take care to pay attention to this aspect of their system design.

Once a key pair has been generated, it is often necessary to convey the public key in an authentic manner to other entities. One way of achieving this authentic distribution is to have the key certified by a trusted Certification Authority within a Public Key Infrastructure.

In many schemes it is desirable for an entity to receive assurance that an elliptic curve public key

is valid or partially valid before they use the public key to, say, verify a signature. This process can help prevent small subgroup attacks and other attacks based on the use of an invalid public key, such as [ABM+03].

As a matter of prudence, a certification authority who issues a certificate for a key pair may want to ensure that the key is not somebody else's. Proof of possession protocols may be well suited for this task, but they do not rule out theft of somebody's private key. In that case, a stronger mechanism, proof of generation, or verifiable key generation, may be desired. It would not seem right for Eve to be able steal Alice's key pair and then get it certified in Eve's name.

Incidentally, the mechanisms for verifiable key generation in which an authority provides input into the key pair, can be also be used as a means for an authority to supplement the entropy of the user's private key. In this case, of course, the authority must be trusted not to abuse this entropy. Similarly, the contribution of the authority must be sent securely to the user, without other parties being able to capture this information. In the situation where the user has only a very weak source of entropy, a properly secure channel is difficult to achieve by purely cryptographic means, so this would have to entail some degree of physical protection of the channel used for entropy delivery. As a further precaution, the information that the authority uses to verify the generation of the key should be kept secret too.

A further discussion of the generation and validation of elliptic curve key pairs can be found in ANS X9.63 [X9.63].

B.2.3 Commentary on Elliptic Curve Diffie-Hellman Primitives

Both elliptic curve Diffie-Hellman primitives in Section 3.3 generate a field element from a private key owned by one entity U and a public key owned by a second entity V in such a way that if both entities execute the primitive with corresponding keys as input, they will both compute the same field element.

The primary security requirement of both the primitives is that an attacker who sees only U and V's public keys should be unable to compute the shared field element. This requirement is equivalent to the requirement that the elliptic curve Diffie-Hellman problem or ECDHP is hard. The ECDHP is stated as follows.

Let E be an elliptic curve defined over a finite field $\mathbb{F}_q$, and let $G \in E(\mathbb{F}_q)$ be a point on E of large prime order n. The ECDHP is, given E, G, and two scalar multiples $Q_1 = d_1G$ and $Q_2 = d_2G$ of G, to determine d_1d_2G.

The ECDHP is closely related to the ECDLP. It is clear, for example, that if the ECDLP is easy then so is the ECDHP. Boneh and Lipton [BL96] show that if the ECDLP cannot be solved in subexponential time, then the ECDHP cannot be solved in subexponential time. Maurer and Wolf [MW96] show that, if a certain auxiliary elliptic curve groups exists, then the ECDHP is almost as hard as the ECDLP. For the 15 recommended NIST curves, Hasse's theorem ensures the existence of a suitable Maurer-Wolf auxiliary elliptic curve group, because the Hasse interval contains a power of two. On the other hand, finding a curve of given order defined over a given finite field appears to be an intractable problem. Muzereau, Smart and Vercauteren [MSV04] constructed other suitable Maurer-Wolf auxiliary groups for most of the NIST curves, however.

Many schemes based on the Diffie-Hellman primitives actually rely on a stronger requirement that the shared field element is not just hard for an attacker to predict, but that the element actually looks random to the attacker. A discussion of this requirement, and its relationship to the ECDHP, can be found in [BV96, Bon98]. The potentially easier problem is known as the *decision Diffie-Hellman problem*.

In certain applications of the ECDH primitive, the private key of one of the entities, say U, is a static private key that does not change over time. In this case, entity U wishes to be sure not just that the ECDHP is a difficult problem, but also that repeated application of the private key operation does not leak information, or if it does, that suitable countermeasures are taken to prevent this leakage. Usually, a one-way key derivation function will prevent such leakage. For a discussion of these issues see [BG04a], which also establishes that in some sense unauthorized parties cannot reproduce the static private key operation of U, unless the associated static private key can be found. See also [Che06].

So far the discussion of the elliptic curve Diffie-Hellman primitives has been germane to both the "standard" primitive and the cofactor primitive. The remainder of this section explains the difference between the two primitives.

A direct assault on the ECDHP is not the only way an attacker might attack schemes that use the Diffie-Hellman primitives. Many schemes which use the primitives are also susceptible to small subgroup attacks [Joh96, LL97, ABM+03] in which an adversary substitutes V's public key with a point of small order in an attempt to coerce U to calculate a predictable field element using one of the primitives. The consequences of these attacks can be severe — in a key agreement scheme, for example, the result can be compromise of a session key shared by U and V, or even compromise of U's static private key.

Two defenses against this attack are recommended here: either validate V's public key and use the "standard" Diffie-Hellman primitive (validating V's public key checks that V's public key has order n and hence prevents the attack), or partially validate V's public key and use the cofactor Diffie-Hellman primitive (using the cofactor Diffie-Hellman primitive with a point in a small subgroup will result in calculation of the point at infinity and hence rejection of the key).

Which of the defenses outlined above is appropriate in a given situation will depend on issues like whether or not interoperability with existing use of the "standard" Diffie-Hellman primitive is desirable (the first defense interoperates and the second does not), and what the efficiency requirements of the system are (the second defense is sometimes more efficient than the first).

Additional information on the elliptic curve Diffie-Hellman primitives can be found in ANS X9.63 [X9.63].

B.2.4 Commentary on the Elliptic Curve MQV Primitive

The elliptic curve MQV primitive generates a field element from two key pairs owned by one entity U and two public keys owned by a second entity V in such a way that if both entities execute the primitive with corresponding keys as input, they will both compute the same field element.

Again, the primary security requirement of the primitive is that an adversary who sees only U and V's public keys should be unable to compute the shared field element. This requirement is

equivalent to the requirement that the ECDHP is hard, as shown in [BG04b, §A.1].

See [LMQ+03] and [Men07] for a discussion of the various security properties of MQV. Additional information on the elliptic curve MQV primitive can be found in ANS X9.63 [X9.63].

B.2.5 Commentary on Hash Functions

Hash functions take inputs from a very large space, and return outputs in a much smaller space. In this specification, it is assumed that a hash function is publicly computable. Hash functions have many applications in cryptography, including:

- Message digesting for digital signatures. Usually the raw form of the digital signature can only sign small amounts of data. Applying a hash function to a message containing a larger block of data, produces a digest which is small enough to apply the raw form of the digital signature algorithm.
- Key derivation functions, for key agreement and key transport. Usually the raw shared secrets arising from key agreement schemes are larger than the keys needed for the corresponding symmetric encryption or authentication schemes. Furthermore, the raw shared secrets sometimes contain structure making them distinguishable for uniformly random bit strings, and perhaps worse, contain information, that if subjected to further abuse, that could leak information about private keys. Key derivation functions use hash function to take a raw shared secret and produce one or more, usually smaller, symmetric keys that appear to be uniformly distributed, and whose exposure does not appear to compromise any static private keys.
- Message authentication. A message authentication code (MAC) is the symmetric key analogue to a digital signature. A MAC may also be thought of as a hash function equipped with a symmetric key. The HMAC construction builds a MAC from an arbitrary hash function, but is especially intended for certain kinds of iterated hash functions, such as SHA-1.
- Random number generation. Cryptographic keys must be generated and held secretly from an adversary. The surest way to secretly generate a key is to generate it randomly. Random values, though, are difficult to obtain in deterministic computing devices. Therefore, environmental data, user input, or specialized non-deterministic devices are necessary to use to generate random noise. Such random noise sources, though, typically produce data that is not uniformly distributed. Hash functions are one way to take the output of such a raw noise, and derive from it something that appears to be more uniformly distributed, thereby making it more useful as a key. Hash functions may also be used as part of a deterministic random bit generators, also known as pseudorandom number generators, which take an initial seed value, such as one obtained from a truly random noise source (perhaps hashed to be appear uniformly random), and then produce a long stream of bits that appear to be distributed uniformly at random.
- Verifiably random curve and point generation. A user of elliptic curve domain parameters generated by a third party may worry that the third party maliciously selected the domain parameters in such a way that the third party can thereby compromise the security of the user. For example, the third party may know of a rare class of elliptic curves in which the

discrete logarithm is significantly easier than average. By deriving the curve as the output of a hash function, such attacks may be thwarted. Loosely speaking, the output of hash generally appears random, and thereby verifying that an elliptic curve is derived from the output of a hash, verifies that the elliptic curve appears random. Therefore, such elliptic curves are called verifiably random.

The five applications of hash function are discussed in greater detail below in the sections where the hash functions are applied. There are many other, often more esoteric, applications of hash functions in cryptography, but these are not covered in the specification, so they will not be discussed further.

Not just any hash function is suitable for use in cryptography. Generally, the hash function must be equipped with certain security properties, or else the cryptographic application to which it is put will become insecure. It is difficult, even for a single application of hash functions, such as ECDSA or HMAC, to determine an exhaustive set of security properties necessary of the hash function in order for the application to be secure. Nevertheless, a non-exhaustive list of security properties can be determined. The following are some security properties of hash function, which have been identified [Bro05b] as necessary for the security of ECDSA.

- Collision resistance. Finding two messages, say M and M', whose hashes are identical, that is, $H(M) = H(M')$ should be infeasible. Such a pair of messages is called a *collision*. Just to be clear, for any fixed hash function, such pairs exist, so technically there exist very efficient algorithms to find a collision. (Accordingly, some more academic definitions of collision resistance are only defined for large families of hash functions. In practice, though, one usually uses a fixed hash function. See [Rog06] for more discussion.) Despite the existence of such collisions, collisions in certain fixed hash functions, especially SHA-1 and SHA-256, have yet to be found. (Collisions have been found in other common hash functions, notably MD5.) Generic algorithms are known to find collisions in any hash function. If the size of the output space is N, then these algorithm find a collision with an expected cost of about a known constant factor of $\sqrt{N}$ times the cost of applying the hash function. Loosely speaking, against such attacks, one can say things like SHA-256 appears to have 128-bit security with respect to collision resistance. In the case of SHA-1, specific algorithms have been discovered [WYY05a, WYY05b] which find collisions more quickly than the generic algorithm, so rather than providing 80-bit security, it is now deemed to provide at most 63-bit security, with respect to collision resistance. The hash functions approved in this specification are taken from other standards, which give collision resistance as a security goal.

- Preimage resistance. For a random value, say e, in the range of the hash functions H, finding a message M which is a preimage of e, that is, such that $H(M) = e$, should be infeasible. A preimage resistant hash function is sometimes also called a one-way function. (A technical difference is sometimes given, however, which is, for H to be called one-way, the value e is not chosen at random from the range of H, but rather chose as $e = H(M')$ for some message M' chosen at random from the domain of H. Here, we are concerned not with this definition, but the former, of preimage resistance.) A generic method to find a preimage is to select N random messages, where N is the size of the output range. This strategy finds a preimage of e with high probability, unless H has a very non-uniform distribution (certain outputs

are much more likely than others). It is conjectured that certain hash functions, including SHA-1 and SHA-256, provided a security level of preimage resistance equal to their output length, in bits. The collision attacks on SHA-1 have so far have not led experts to suggest that SHA-1 has less that 160-bit security against preimage attacks. If the hash function has certain uniformity properties, then it can proved to have at least as much preimage resistance as collision resistance. [Sti06].

- Second preimage resistance. For a random message M, finding a second message M' such that $H(M) = H(M')$ should be infeasible. This property, and the associated problem, is parameterized by the probability distribution of the random message M. Typically, the distribution over which one wants to define the property are useful messages that one is may possibly want to sign, in certain situations. Collision resistance implies second preimage resistance, but it seems reasonable to hope for twice as many bits of security for second preimage resistance than for collision resistance. The collision attacks on SHA-1 [WYY05a, WYY05b] do not seem yet to carry over to second preimage attacks.

- Zero preimage resistance. For given elliptic curve domain parameters including the prime order n of the base point G, it should be infeasible to find a message M such that $H(M) \equiv 0 \pmod{n}$. This security property is necessary for the security of ECDSA, but is otherwise *not* a standard security property expected of hash functions. One of the reasons for choosing an elliptic curve verifiably at random is to prevent an attacker from choosing some message M to forge, and then selecting elliptic curve domain parameters such that $n = H(M)$, which may be possible to do using the complex multiplication method of generating elliptic curves. This type of domain parameter attack, originally conceived by Vaudenay in the context of DSA, can be regarded as a variation of the attack of finding a preimage of zero. Once n has been fixed, then finding a zero preimage is a problem similar to finding a preimage of a random element. The two problems, though, are strictly incomparable. It could be easy to find preimages of random hash values, but there may not even exist any preimages of zero. Conversely, it may be easy to find a preimage of zero, but difficult to find preimages of random hash values. Much like collision resistance, there is a subtlety with defining zero preimage resistance in the sense that a very efficient algorithm may exist to solve: namely the algorithm that simply produces the preimage. So, when we speak here of zero preimage resistance, we do not mean that an efficient algorithm does not exist, but rather that all known algorithms to find a preimage of zero are infeasible.

- Rarely zero. This is the same as zero preimage resistance, except that instead of it being infeasible for the adversary to find a message M with $H(M) \equiv 0 \pmod{n}$, there should be negligible probability that a random message M is such that $H(M) \equiv 0 \pmod{n}$. This security property is not defined in terms of an adversary. This security property is strictly implied by zero preimage resistance. It is also implied by collision resistance. The reason to consider this rather weak security property separately is that it is necessary for ECDSA to satisfy some basic but rather weak security goals, whereas, collision resistance of the hash function is not known to be necessary for ECDSA to satisfy the same basic security goals, such as universal forgery against no-message attacks.

For other applications of hash functions, especially applications to random number generation, further security properties are likely to be necessary. For example if a random input is given to

the hash function, then the output should look random too. If a bit of the hash function output were fixed, or biased, then the hash function would likely not be suitable to build random number generators. A fixed bit in the hash function only slightly diminishes the five security properties above. Therefore, the list above is not an exhaustive list of required security properties of hash functions.

On the other hand, in certain applications of hash functions other than ECDSA, not all the above security properties are known to be necessary. For example, collision resistance does not appear to be a necessary condition for the security of the other applications of hash functions in this standard. That is to say, there is no known method that leverages only a collision in the hash function as a means to attack these other hash applications (MAC, KDF, random number generator, or verifiably random curve generation scheme). The necessary security properties of the hash function to ensure the security of these other applications is not well understood.

Further properties of hash functions are used in the area of *provable security*, also known as *Foundations of Cryptography*, which is the ability to prove security properties of a scheme based upon properties of the components that make up the scheme. It is worth mentioning two additional properties of hash function which enable proofs of desirable security properties of ECDSA [Bro05b]. These properties do not appear to be necessary to ensure the security of ECDSA.

- Uniformity (Smoothness). There is a distribution of messages whose hashes can be efficiently computed, such that the preimage of each hash value is large in the sense that the probability of correctly guessing a random element in a set that large is negligible, and exhaustively searching a set that large is infeasible; and the hash values of messages with this distribution are infeasible for an adversary to distinguish from random values in the range of the hash function. Loosely speaking, we may think of this security property as *random-in random-out* or as a kind of *pseudorandomness* property. This property, together with collision resistance, imply preimage resistance. This security property is rather mild in the sense that it is easy to construct examples of hash functions that can be proven to have this property. It would be somewhat surprising if hash functions like SHA-1 or SHA-256 did not have this property. (It could fail, for example, if some linear combination of the bits or bytes of SHA-1 was fixed.) There are no known attacks on ECDSA based on the event that the hash function fails to be uniform. The main reason to consider this security property for ECDSA is that it allows for a proof of security in the generic group model for the elliptic curve [Bro05a].

- Random oracle model. This means that, usually only in the context of a proof of security, one models the hash function by a random function which the adversary may access only via an oracle. In practice, a hash function is not a random function, and further the adversary has unlimited access to the hash function. As such, this is not an obtainable security property. Therefore, the terminology *random oracle model* is used to reflect that one is only working in a *model* of reality. A real random oracle does not exist, so it can only serve as a model for a real function. Despite this limitation, it is generally believed that a proof in the random oracle model provides some assurance. Generally, in the random oracle model all of the other security properties listed above of hash functions hold true. Very loosely speaking, whereas the uniformity property may be thought of as *random-in random-out*, this property may be thought of as *anything-in random-out*.

There is plentiful literature on the security properties of hash functions. This section has only attempted to summarize some of the theory that is most relevant to elliptic curve cryptography, especially the ECC schemes defined in this standard. That the emphasis is mainly on the impacts of hash function security on ECDSA, is primarily because more study has been done in this area, and secondarily because ECDSA appears to depend more critically upon the hash function than, say, ECMQV does.

B.2.6 Commentary on Key Derivation Functions

In this standard, key derivation functions are used to take a raw shared secret, either from an ECDH or an ECMQV primitive, and produce a symmetric key which is then used as part of a key agreement scheme, encryption scheme or key transport scheme. Compared to hash functions, much less work has been done on the security of key derivation functions. There are several informal purposes for applying the key derivation function to the raw shared secret before using it as a symmetric key.

- The raw shared secret may be too long for the desired symmetric key algorithm (either encryption or MAC) intended to be used. As such, the key derivation function has to provide a compression utility.
- The raw shared secret may be too short if multiple symmetric keys are desired. As such, the key derivation function has to provide an expansion utility.
- The raw shared secret may have some mathematical properties that, in the event of its exposure, may be exploitable for further attacks. In certain circumstances the symmetric key may be exposed, because of its heavy use and sharing with other parties. As such, the key derivation function should be a one-way function (or preimage-resistant) so that exposure of the derived symmetric key does not expose the raw shared secret.
- The raw shared secret may have some mathematical structure makes it distinguishable from a random bit string. Generally, symmetric keys are expected to be indistinguishable from random bit strings. As such, the key derivation function has to provide the utility sometimes called randomness extraction, which is to take as input a random but biased value and produce as output a value that appears to have a uniformly random distribution.

B.2.7 Commentary on MAC Schemes

Considerable research has gone into the design of message authentication codes, including the schemes HMAC and CMAC allowed in this standard. These are standardized in external NIST standards, so no further comment is provided here.

B.2.8 Commentary on Symmetric Encryption Schemes

Considerable research has gone into the design of symmetric encryption schemes, including the schemes Triple-DES and AES, and the various block cipher modes, allowed in this standard. These are standardized in external NIST standards, so no further comment is provided here.

When using ECIES, some exception are made. For the CBC and CTR modes, the initial value or initial counter are set to be zero and are omitted from the ciphertext. In general this practice is not advisable, but in the case of ECIES it is acceptable because the definition of ECIES implies the symmetric block cipher key is only to be used once.

One option of ECIES is to use XOR as a symmetric encryption scheme. This is similar to using the CTR mode of a block cipher, in that a key stream is generated from the raw shared secret and then is XORed with the plaintext. The main difference is how the key stream is generated. In the XOR mode of ECIES, the key stream is provided by the KDF, whereas with the CTR mode of a block cipher it is generated via KDF, a counter and a block cipher. However, when the KDF is used to generate a long key stream it operates very similarly to the CTR mode, by hashing a counter with a secret.

It cannot be overemphasized that the key streams obtained from CTR and XOR modes must be used as one-time pads only. In particular, re-use of the key stream is likely to leak considerable information about the messages encrypted under the re-used key streams.

B.2.9 Commentary on Key Wrap Schemes

Key wrap schemes are essentially specialized symmetric encryption schemes whose plaintext contain keying information. In one respect, key wrap schemes need to be more robust than general symmetric encryption because their content contains inherently sensitive information whose loss could lead to loss of other information. In an opposite respect, however, key wrap schemes are less sensitive to losses of small amount of information, since generally a small leakage of information of a secret key (whether symmetric or asymmetric) does not *a priori* lead to a compromise of the usage of the key. A third respect in which the functionalities may be distinguished is that the input to key wrap is generally of a shorter, bounded length, whereas a general purpose symmetric encryption scheme typically must be able process very large messages.

B.2.10 Commentary on Random Number Generation

Random numbers are most important in elliptic curve cryptography for generation of the elliptic curve private keys. Of utmost importance is the secrecy of the private key. It should be infeasible for an adversary to exhaustively search through all possible values of a private key. The general accepted best measure against exhaustive search is called minimum entropy, or min-entropy for short. The min-entropy of the private key is the base two logarithm of the maximum probability, from the adversary's perspective, of any value of the private key. For example, if the maximum probability of any value for the private key is 2^{-80}, then the private key can be said to have 80 bits of min-entropy.

For a private key of length 160 bits, the min-entropy is at most 160 bits. The min-entropy is maximized precisely when each private key is equally likely. However, in practice, getting computer systems to generate random numbers with sufficient entropy, let alone uniform distribution, is a difficult task.

Note that min-entropy is considered more suitable than Shannon entropy for cryptographic applications. Consider a 160 bit private key, whose distribution is not perfectly uniform. One may

suspect that 80 bits of Shannon entropy would be sufficient to make sure the key resists exhaustive search. This is false. Although, Shannon entropy is an excellent measure for coding theory and data compression, it is not as suitable for cryptography. Consider a pathological random number generator such that one value of the key has probability 1/2 and all others have probability of about 2^{-160}. The Shannon entropy of this random number generator is 80 bits. For cryptography, however, this pathological random number generator is insecure. An adversary has probability of 1/2 of guessing the private key. The rest of the time, again with probability 1/2, the private key will be infeasible for the adversary to guess (that is, exhaustively search). This pathological random number generator only provides 1 bit of min-entropy even though it provides 80 bits of Shannon entropy. A theorem of Renyi states that min-entropy is always less than Shannon entropy. Given that practical concerns dictate that random number generators cannot be made to perfectly uniform, we deem that min-entropy, not Shannon entropy, is the correct security measure for non-uniform random number generators.

In practice, random number sources, such as hard disk read times, may only provides small amounts of entropy compared to what is needed for a given security level. If the randomness sources are truly independent, then the random values can be combined and the min-entropy of the combination is the sum of the min-entropy of the parts. In this way, a sufficient amount of entropy can be accumulated. In practice, it is difficult to know definitively how much min-entropy an individual randomness sources provides, and it is difficult to be sure that individual randomness sources are independent. Nevertheless, the principles elucidated above give a general strategy to accumulate sufficient min-entropy.

Generally, computer systems are designed not to be random. Thus it is intrinsically difficult to find randomness sources. Customized hardware, such as noisy diodes or even sources based on radioactive decay or quantum effects, may provide very reliable sources of entropy. Common hardware components, such as hard disks, that have performance variations can also provide some entropy. More generally, in complex operating systems, the timings of certain processes may actually provide some randomness. Similarly, data stored on a hard drive, such as user files, varies over time, and is individual to a user, and as such may provide some entropy. Unfortunately, these sources may not vary enough over time often to be useful for cryptography, and furthermore may not be sufficiently secret enough either. User inputs, such as keyboard strokes or mouse movements, may also provide some randomness.

Entropy alone, however, is not necessarily sufficient for elliptic curve private keys. Private keys that have sufficient entropy to resist exhaustive search can be very insecure for use in ECC. For example, when using a 160-bit curve, a uniformly random private key between 1 and 2^{80} will resist exhaustive search, but the small size of the private key means that it can found from the public key with cost of about 2^{40} elliptic curve group operations using Pollard's lambda algorithm for finding discrete logarithms.

Furthermore, for ECDSA ephemeral private keys, much smaller amounts of bias can lead to attacks. Howgrave-Graham and Smart describe an attack where, if the attacker can learn the five most significant bits of the ECDSA ephemeral private key in a few hundred signatures, then the attacker can compute the private key. Nguyen and Shparlinksi [NS03] describe an improvement of this attack. Bleichenbacher [Ble01] describes an attack exploiting even less bias. His attack works if, for example, an ECDSA ephemeral private key over a 160-bit elliptic curve is generated in the range 1 to $3(2^{158})$. If an adversary can collect about four million ECDSA signatures generated

with these biased ephemeral private keys, then the adversary can determine the associated static private key.

Bellare, Goldwasser and Micciancio [BGM97] describe an attack on DSA, which could potentially be applied to ECDSA too. If the random number generator that signer uses to generate ECDSA ephemeral private keys is a linear congruential generator, then the attacker can determine the signer's private key after seeing just a few signatures. This attack suggests that, not only must each ephemeral private key be free of bias, but moreover there must not be any strong correlations between successive ephemeral private keys.

It therefore makes most sense to generate all elliptic curve private keys with a random number generator that (a) has sufficient entropy to resist exhaustive guessing attacks, and (b) has outputs indistinguishable from independently and uniformly random private keys. The latter is not strictly necessary, since the attacks of Bleichenbacher only work for certain types of bias. Nevertheless, the latter is believed to be easily achievable using random number generators based on hash functions, block ciphers, or even elliptic curves.

Since the cost of gathering min-entropy is high, it is generally consider best to seed a pseudorandom number generator with a sufficient amount of entropy, and optionally to provide it with any additional entropy that can be gathered during its lifetime. This is a robust design, in that if the real-time entropy should fail, the random number generator still provides pseudorandom numbers.

Random number generators, however, can be captured by adversaries. It is important that, if this is to happen, that the adversary cannot determine previous outputs of the random number generator. This is called backtracking resistance. Modern designs of random number generators incorporate this security feature by means of one-way functions. Older designs generally did not guard against this threat, and as such, are not recommended in this standard and most other newer standards.

Another kind of active attack is when an adversary somehow learns or influences the state of a random number generator. If the random number generator has a feature whereby it gathers and uses an additional source of entropy, then provided that sufficient entropy has been gathered, its output should become secure against an adversary who previously learned the state. This is called prediction resistance, and is considered an optional feature of most modern random number generators.

The elliptic curve random number generator has backtracking resistance and optional prediction resistance. It is proven [BG07] to provide outputs that are indistinguishable from random outputs not as bit strings, but rather as values derived from the x-coordinates of elliptic points. Ongoing research suggests that this feature should make them suitable for use as elliptic curve private keys.

B.2.11 Commentary on Security Levels and Protection Lifetimes

This standard follows NIST and ANSI in fixing five security levels: 80, 112, 128, 192, and 256. Future revisions of this standard may amend this.

Although lower, intermediate, and even higher security levels are possible, for greater interoperability, these five are fixed. Other sizes of elliptic curves are assigned to the highest of these five security level with which they are consistent.

The protection lifetimes of security levels have been extrapolated from similar NIST recommendations. The extrapolations are also loosely based on a simple assumption similar to Moore's law: computing power will grow by a factor of about 2^{16} every decade. Therefore, the minimum adequate security level must increase by 16 bits every 10 years. Future revisions of this standard may amend this.

In this model, the base case is that 80 bits of security is adequate until 2010. The next security level, 112 bits, which is 32 bits higher, is adequate for another 20 years, so, until 2030. These are also the NIST recommendations. It may be the case today that some applications require protection beyond 2030. For example, copyright currently lasts longer than 20 years, so if cryptography is to be useful in the protecting copyright, then a security level higher than 112 bits may make sense to use in 2010. The extrapolations for the next three security levels, 128, 192 and 256 bits are that they are good until the years 2040, 2080 and 2120, respectively.

Since this standard mandates five security levels, once one of the first four of the five years given above is passed, one should move to the next higher security level. More precisely, if one needs protection beyond one of the first four of the five years given above, one needs to move to the next higher security level. For example, to obtain protection beyond 2010, the security level of 112 bits is needed. This creates some artificial jumps in the required security level, but as noted above, this is to improve interoperability.

These recommendations do not provide protection beyond 2120, because the highest security level is 256 bits. It is, however, unlikely that anybody today will need protection beyond 2120. If they do, then users of this standard are suggested to use, larger elliptic public key sizes, extrapolating the security level according the pattern above. Unfortunately, they could not rely on the symmetric algorithms allowed in this standard, and would therefore need to resort to some other symmetric algorithms.

B.3 Commentary on Section 4 — Signature Schemes

This section provides commentary on Section 4 of the main body of this document.

B.3.1 Commentary on the Elliptic Curve Digital Signature Algorithm

The ECDSA is a signature scheme with appendix based on ECC. It is designed to be existentially unforgeable, even in the presence of an adversary capable of launching chosen-message attacks. Vanstone [Van92] was the first to propose to develop an elliptic curve analog of the U.S. government's Digital Signature Algorithm (DSA) [186].

The ECDSA was chosen for inclusion in this document because it is widely standardized in, for example, ANS X9.62 [X9.62b], IEEE 1363 [1363], and ISO 15946-2 [15946-2]. Its widespread standardization, together with its close relationship to DSA, means that both specification details and implementation details have been carefully scrutinized. Standardization has also led to the provision of valuable tools such as the Cryptographic Module Validation Program (CMVP) ECDSA validation system, whereby implementers can get accredited implementation testing laboratories to check that their code is free from errors.

The features of ECDSA were considered to outweigh, at the time that the previous edition of this standard [SEC 1] was being finalized, the features of other candidates like the Schnorr scheme [Sch91] which was shown to be provably secure in the random oracle model based on the ECDLP in [PS96], or the Nyberg-Rueppel scheme [NR93, NR96] which avoids the need for modular inversion during signature generation and verification and can offer slightly smaller signature sizes through its message recovery capability. Subsequently, the security of ECDSA has been proved in the generic group model [Bro05a] and under a variety of other assumptions [Bro05b]. Some potential limitations on the provable security of signature schemes like ECDSA were found by Paillier and Vergnaud [PV05]. Signature schemes providing partial message recovery, such as Pintsov-Vanstone signatures [PV00] proven secure in [BJ01] and standardized in [X9.92], may be included in a future SECG standard.

There are a number of known cryptographic attack methods on ECDSA. The specification of ECDSA in this document includes provision for preventing all of these attack methods. Nonetheless implementers should be aware of the attacks and monitor future advances. The attacks illustrate the importance of ECDSA implementations performing all the security checks specified in the main body of this document. The following is a list of some of the known attack methods:

- Attacks on the ECDLP. The security of ECDSA relies on the difficulty of the ECDLP for the elliptic curve domain parameters being used — otherwise an attacker may be able to recover U's private key from U's public key and thereafter use this information to forge U's signature on any message.

- Attacks on the elliptic curve semi-logarithm problem (ECSLP), introduced in [Bro01, Bro05b]. A *semi-logarithm* of point P to the base G is a pair (t, u) of integers such that $P = f(u^{-1}(G + tP))$, where f is the function used in ECDSA that converts the ephemeral public key point $R = kG$ into the signature part r, so that $r = f(R)$, which essentially consists of taking the x-coordinate of R and reducing it modulo n. An algorithm to compute a semi-logarithm can be used to compute an ECDSA signature. It is not known whether the ECSLP is significantly easier than the ECDLP (though it is obviously no harder). Some evidence of a gap between the problems may be exhibited by [PV05].

- Attacks on key generation. Key generation is involved in both the key deployment procedure and the signing operation of ECDSA. Secure random or pseudorandom number generation is required during key generation to prevent, for example, U from selecting a predictable private key. Insecure random and pseudorandom number generators are perhaps the most common cause of cryptographic attacks on cryptographic systems. Note that both the static private key d and each per-signature ephemeral private key k must be chosen securely. An attacker who learns a single k can recover d, and thereafter forge signatures at will. Furthermore, various results [HGS01, NS03, NNTW05] have shown that a small amount of bias in k can also gradually leak the private key d.

- Attacks on the hash function. The hash function used by ECDSA during the signing operation and the verifying operation must possess a number of properties such as one-wayness and collision resistance. Otherwise if the hash function is not, say, collision resistant, an attacker may be able to find a collision (M_1, M_2) and forge U's signature on M_2 after persuading U to sign M_1. The five necessary security properties for the hash function used in ECDSA listed

in [Bro05b] are: rarely zero, zero-resistant, 1st-preimage resistant, 2nd-preimage resistant, and collision resistant.

- Attacks on the ECDSA conversion function. One of the steps in ECDSA is to convert an ephemeral public key $R = kG$ to an integer $r = f(R)$. This conversion function essentially entails taking the x-coordinate of R and reducing it modulo n, where n is the order of the base point G. For ECDSA to be secure, it is shown in [Bro05b] that this function needs to be almost bijective, which essentially means that an attacker cannot find an r for which a random R has non-negligible probability of satisfying $r = f(R)$. For an elliptic curve with a small cofactor, it is simple to show that the conversion function is almost bijective.

- Attacks based on invalid domain parameters. The security of ECDSA relies on U using valid domain parameters because, for example, invalid domain parameters may be susceptible to the Pohlig-Hellman attack [PH78]. Entity U should therefore receive assurance that the elliptic curve domain parameters used are valid. Entity V may also desire to check that the elliptic curve domain parameters are valid to prevent attacks like those described in [BWM99, CH98] and to mitigate against the possibility of a repudiation dispute in which U denies liability because U was using invalid domain parameters.

- Attacks based on invalid public keys. It may be desirable for V to check that U's public key is valid to prevent, for example, attacks like those described in [BWM99, CH98]. Another class of attack to avoid is described in [ABM+03]. Another reason V may wish to check that U's public key is valid is to mitigate against the possibility of a repudiation dispute in which U denies liability because U was using an invalid public key.

- Vaudenay's attack. Vaudenay [Vau96] showed that ECDSA is susceptible to attack if an attacker is able to persuade entity U to use elliptic curve domain parameters with base point order n chosen by the attacker and satisfying $\lceil \log_2 n \rceil \leq 8(hashlen)$. This attack can be prevented, for example,

  - through exclusive use of elliptic curve domain parameters generated by U or by some trusted party,
  - through use of verifiably random elliptic curve domain parameters or elliptic curve domain parameters from a small well-known family of parameters like parameters associated with Koblitz curves, or
  - through use of parameters with $\lceil \log_2 n \rceil > 8(hashlen)$.

- Duplicate signatures. Stern, Pointcheval, Smart and Malone-Lee [SPMLS02] showed how to create duplicate signatures, as defined below, for ECDSA. A malicious signer can find two messages for which a single signature is valid. This is not regarded as a forgery attack, because no signature has been created without the use of a private key (see also [Bro05b]). The malicious signer may try to repudiate the signature on one of the messages. An argument for repudiation entailing that some third party caused the duplicate signature to occur seems to presuppose the existence of genuine forgery attack. Because such attacks are unknown, the balance of the probabilities falls to malice by the signer, since the signer has access to the private key and therefore much greater ability to generate signatures. Over and above these general concerns, this duplicate signature attack has the added deficit that it exposes

the signer's private key. Because the private key is determined by the signature and the two messages signed, the probability that the signer had generated the private key in an honest manner is negligible, and one can deduce almost certainly that the private key was deliberately generated in order to launch this attack. To repudiate the signatures, a signer would therefore have to assert that some third party generated the signer's private key, which contradicts the usual assertion that signers must make for non-repudiation: namely, that the signer has generated the private key and not revealed it to anybody else.

- Malleable signatures. If signature (r, s) is a valid signature for a given message, then so is $(r, -s \bmod n)$. This is not regarded as a forgery because the same message is signed in both cases (see also [Bro05b]).

Of course a variety of non-cryptographic attacks on ECDSA are also possible, and implementers should take precautions to avoid, for example, "implementation attacks" such as fault-based attacks [BDL97], power-analysis attacks [KJJ99], and timing-analysis attacks [Koc96].

The operation of ECDSA involves selection by implementers of a number of options. These choices will typically be made based on concerns like efficiency, interoperability, and on security issues like those outlined above. In particular, some of the choices involved are selection of elliptic curve domain parameters, selection of a hash function, and selection of parameter and public key validation methods. Selection of elliptic curve domain parameters will likely involve consideration of issues like those discussed in Sections B.1 and B.2, and selection of parameter and public key validation methods will likely involve consideration of issues like those discussed above.

Implementers of ECDSA may wish to use the methods in [ABG+05] to accelerate the speed of verification. Signers, or a third party, can help verifiers do this by choosing between one of the form (r, s) and $(r, -s \bmod n)$ in a canonical manner, so that the verifier can recover correctly the elliptic curve point R corresponding to r that allows faster verification of the signature as described in [ABG+05]. To further accelerate verification, the signer, or a third party, may also provide with the signature additional information allowing the verifier to compute R from r more rapidly.

The alternative verifying operation would typically be used by a certification authority to more efficiently verify a certificate that it itself issued. Because this operation outputs the validity of a pair of integers (r, s) as an ECDSA signature, which is publicly available information, it is difficult to see how the output could be used to aid an adversary in any way. Of course, if the verifier has extra information about R, then it may be usable to even further accelerate verification.

The public key recovery operation is useful in its own right for situations in which the signer wishes to use less bandwidth. For example, a signer Alice may send her signature (r, s) and her certificate to Bob, but with this operation, she can omit her public key Q from the certificate. Bob can recover Q from (r, s) and then verify the certification authority's signature on the public key. Note that recovery of Q from the signature (r, s) does not guarantee the validity of (r, s), since generally any signature will give rise to some Q. Verification may instead be regarded as implicit, however, because once the authenticity of Q is confirmed, then the signature (r, s) can be regarded as valid. So, not only does this reverse the usual order of validation from first Q then (r, s) to first (r, s) then Q, but the intermediate validation is implicit.

The self-signed signature operation is another application of the public key recovery operation. In turn, the self-signature operation is applicable to verifiable key generation, which may be used

to demonstrate that nothing untoward is being done with a key pair. Self-signed signatures may not have very significant applications in their own right, but it is worth discussing their security, specifically with regard to their suitability to verifiable key generation. The main security objective of a self-signature is that, given a public key Q, it is infeasible for anybody to find a self-signed signature (r, s) that recovers to Q. To see why this may be true, we model the hash function to be secure in the sense of a random oracle. Because the signature is part of the input to the random oracle, the output, which is e is essentially random. Therefore, the public key Q recovered from (r, s) is essentially random. Therefore, it is infeasible to cause the recovered Q to land on any specific pre-existing choice for Q. In other words, the function from (r, s) to Q appears to be a one-way function.

Additional information on ECDSA, including an extensive security discussion, can be found in the standards ANS X9.62 [X9.62b] and IEEE 1363 [1363], and in the papers [Bro05a, Bro05b, JMV01]. Test vectors for ECDSA can be found in ANS X9.62 [X9.62b].

B.4 Commentary on Section 5 — Encryption Schemes

This section provides commentary on Section 5 of the main body of this document.

B.4.1 Commentary on the Elliptic Curve Integrated Encryption Scheme

The ECIES is a public-key encryption scheme based on ECC. It is designed to be both semantically secure and plaintext-aware in the presence of an adversary capable of launching chosen-plaintext and chosen-ciphertext attacks. It was proposed in [BR97, ABR01b, ABR01a].

Note that the specification of ECIES here differs slightly from the description in [ABR01b] where it is mandated that R is included in the input to the key derivation function. Absence of R can affect the malleability, or adaptive chosen-ciphertext security, see [Sho01], but ECIES without R only suffers from *benign malleability*, which is arguably not at all a concern. Nevertheless implementations may of course choose to include R in the input to the key derivation function to achieve complete alignment with [ABR01b].

The ECIES was chosen for inclusion in this document because it offers an attractive mix of provable security and efficiency. It was proven secure based on a variant of the Diffie-Hellman problem in [BR97, ABR01b, ABR01a, Den05, Sma01, CS01]. It is as efficient as or more efficient than comparable schemes. The dominant calculations involved in encryption are two scalar multiplications, and the dominant calculation involved in decryption is a single scalar multiplication. ECIES is also standardized in ANS X9.63 [X9.63] and IEEE 1363A [1363A] and is under consideration in ISO [18033-2].

The features of ECIES outlined above were considered to make it the most attractive ECC-based public-key encryption scheme for standardization at the time of the previous editions [SEC 1] of this standard. In particular, of the other possibilities, the elliptic curve analog of traditional ElGamal encryption [ElG85] does not offer security against chosen-ciphertext attacks, while the elliptic curve analog of the Cramer-Shoup encryption scheme [CS98] offers similar security properties but is considerably less efficient.

There are a number of known attack methods on ECIES. The specification of ECIES in this document includes provisions for preventing all these attack methods. Nonetheless implementers should be aware of the attacks and monitor future advances. The attacks illustrate the importance of ECIES implementations performing all the security checks specified in the main body of this document. The following is a list of some of the known attack methods:

- Attacks on the ECDLP or ECDHP. The security of the ECIES relies on the difficulty of the ECDLP and ECDHP for the elliptic curve domain parameters used — otherwise an attacker who sees an encrypted message sent from U to V may be able to recover the shared secret value z from R and Q_V, and thereafter use this information to discover the message.

- Attacks on key generation. Key generation is involved in both the key deployment procedure and the encryption operation of ECIES. Secure random or pseudorandom number generation is required during key generation to prevent, for example, V selecting a predictable private key. Insecure random and pseudorandom number generators are perhaps the most common cause of cryptographic attacks on cryptographic systems.

- Attacks on the symmetric encryption scheme. The symmetric encryption scheme used by the ECIES need only possess only mild security properties to ensure the security of ECIES. (That is why the XOR encryption scheme may be used by ECIES.) Nonetheless severe compromise of the symmetric encryption scheme may result in leakage of information about encrypted messages.

- Attacks on the MAC scheme. As was the case for the symmetric encryption scheme, the MAC scheme used by ECIES need only possess only mild security properties to ensure the security of ECIES. Nonetheless severe compromise of the MAC scheme may enable an attacker to launch a chosen-ciphertext attack on ECIES.

- Attacks on the key derivation function. The key derivation function used by ECIES must possess a number of properties to ensure the security of ECIES. If, for example, an attacker is able to predict some bits of the output of the key derivation function, or if portions of the output of the key derivation function are correlated in some way, an attacker may be able to learn some information about encrypted messages. These concerns provide some motivation for the use of the TDES or AES symmetric encryption scheme rather than the XOR symmetric encryption scheme when using ECIES to convey long messages since this choice minimizes the amount of output the key derivation function is asked to produce.

- Attacks based on the use of invalid domain parameters. The security of ECIES relies on V using valid domain parameters because, for example, invalid domain parameters may be susceptible to the Pohlig-Hellman attack [PH78]. Entity V should therefore receive assurance that the elliptic curve domain parameters used are valid.

- Attacks based on the use of invalid public keys. When the ECIES is used with the standard elliptic curve Diffie-Hellman primitive, V should check that the public key R is valid to prevent Lim-Lee style small subgroup attacks [Joh96, LL97, ABM+03] which may allow an attacker to learn some bits of V's private key. Similarly, when ECIES is used with the cofactor Diffie-Hellman primitive, V should check that the public key R is partially valid to

prevent the possibility of similar attacks. (The cofactor Diffie-Hellman primitive is designed specifically to enable efficient prevention of small subgroup attacks.)

- Attacks based on the absence of R from the key derivation function input, introduced in [Sho01, §15.6.1]. If the optional input R to the key derivation function is omitted, then an attacker may be able to substitute R with another value R'. We assume here that entity V validates R', or at least partially validates R' when using the cofactor Diffie-Hellman primitive. The substitution $R' = -R$ gives a fully valid ephemeral public key point R' with the same x-coordinate as R. Because the encryption and MAC keys are derived from the x-coordinate of the shared secret, the substituted point R' will not affect the derived key, and thus the ciphertext will remain valid and the plaintext will remained unchanged. This phenomenon has been called *benign malleability*, and is generally deemed harmless, because even though the ciphertext has been modified, the plaintext has not been modified. Formal definitions of the security of public-key encryption schemes can be adapted to regard benign malleability as acceptable, and then the existing security proofs for ECIES appear to apply to this modified form of security when R is omitted. It has been noted in [Sho01, §15.6.1], however, that omission of R appears to loosen the security bounds of certain security proofs. In the case of partially valid R, an attack can also use $R' = \pm R + S$ where S is a partially valid point of order dividing the cofactor h. Under cofactor multiplication, this modified R' does not affect the derived key. Again, this specification allows for entities U and V to include R in $\mathit{SharedInfo}_1$ as an optional mechanism to address any concerns with benign malleability.

- Attacks based on ambiguity of the MAC input, introduced in [Sho01, §15.6.3]. The input to the MAC is $EM \,\|\, [\mathit{SharedInfo}_2]$. It is recommended that entities U and V agree on a format for $\mathit{SharedInfo}_2$ that prevents any ambiguity in where EM ends and $\mathit{SharedInfo}_2$ begins. If such ambiguity is allowed, however, an adversary can substitute EM with EM' and $\mathit{SharedInfo}_2$ with $\mathit{SharedInfo}'_2$. Provided that $EM \,\|\, \mathit{SharedInfo}_2 = EM' \,\|\, \mathit{SharedInfo}'_2$, then entity V will accept the modified ciphertext as valid. Clearly, this means either that EM' is a prefix of EM, so that $EM = EM' \,\|\, EM''$, and then $\mathit{SharedInfo}'_2 = EM'' \,\|\, \mathit{SharedInfo}_2$ so that $\mathit{SharedInfo}_2$ is a suffix of $\mathit{SharedInfo}'_2$, or that $\mathit{SharedInfo}'_2$ is a suffix of $\mathit{SharedInfo}_2$, so that $\mathit{SharedInfo}_2 = \mathit{SharedInfo}''_2 \,\|\, \mathit{SharedInfo}'_1$ and $EM' = EM \,\|\, \mathit{SharedInfo}''_2$. In the former case, entity V will decrypt the ECIES ciphertext to obtain a plaintext that is shorter than the plaintext that U sent; more precisely, it will be a prefix. In the latter case, entity V will decrypt a plaintext that is longer than the one U sent; more precisely, it will be concatenated by a more or less random-appearing appendix. Note that this attack, while a true malleability attack and definitely far more severe than a benign malleability attack described above, does not impart the attacker with full power that is often ascribed to malleability attacks when stating their importance. In particular, the attacker can modify part of the message, say from "Bob" to "Eve", even if he already knows the value of and location in the plaintext of the part she wants to modify. Nevertheless, it is not unimaginable that considerable chaos, if not harm or benefit to the attacker, can be created by the attacker truncating plaintext suffixes, or appending random data to plaintexts.

- Attacks based on truncating and modifying the plaintext, introduced in [Sho01, §15.6.4]. These attacks only apply when both (a) the symmetric encryption used is the XOR mode (using the KDF output as a stream cipher) and (b) the MAC key is taken from the end of

the KDF output and not the beginning. This standard allows ECIES to be used in this mode only for backwards compatibility. Suppose the plaintext has the form $M = M_1 \| M_2 \| M_3$, where M_2 has the length of the MAC key and is known to the attacker. The attacker can then modify the ciphertext to be a valid encryption of $M_1 \oplus \Delta$ for any Δ of the attacker's choice. This attack is preventable by several means, including: (a) not using XOR for encryption, (b) drawing the MAC key from the beginning of the KDF output, (c) fixing the length of the message, and (d) ensuring attackers do not get to know M_2. When ECIES is used for key transport, in which case M is a symmetric key, then M is expected to have a fixed length and to be a secret value, so this attack would not be feasible. It should be borne in mind that in this attack, the attacker does not learn the plaintext directly. Rather, the attacker modifies the plaintext. An attacker may gain from this in various ways. The recipient may take some action that is more likely to benefit the attacker then the sender. Or, the recipient may react in some way that the attacker can use to learn additional information about the contents of the original plaintext M. If possible, users of ECIES should not use ECIES in the backwards compatibility mode, because of this attack. If however, users of ECIES need to use ECIES in the backwards compatibility mode, users should take measures to thwart these attacks, especially fixing the length of M, or else investigate whether the attacks are irrelevant for the application at hand, such as key transport.

Of course a variety of non-cryptographic attacks on ECIES are also possible, and implementers should take precautions to avoid, for example, "implementation attacks" such as fault-based attacks [BDL97], power-analysis attacks [KJJ99], and timing-analysis attacks [Koc96].

The operation of ECIES involves selection by implementers of a number of options. These choices will typically be made based on concerns like efficiency, interoperability, and on security issues like those outlined above. In particular, some of the choices involved are selection of elliptic curve domain parameters, selection of a key derivation function, selection of a symmetric encryption scheme and MAC scheme, selection of the standard or cofactor Diffie-Hellman primitive, selection of parameter and public key validation methods, and selection of appropriate data to include in $SharedInfo_1$ and $SharedInfo_2$. Selection of elliptic curve domain parameters will likely involve consideration of issues like those discussed in Sections B.1 and B.2. Selection of a symmetric encryption scheme will likely be influenced by the length of messages which are going to be encrypted and the amount of memory available. (When the TDES encryption scheme is used, messages can be passed into the encryption operation a piece at a time, whereas when the XOR encryption scheme is used to encrypt variable length messages, the length of the message must be known before MAC computation can begin.) Selection of the HMAC–SHA-1–160 scheme or the HMAC–SHA-1–80 scheme will likely be influenced by the need to balance the added security offered by the former against the bandwidth savings offered by the latter. Selection of the standard or cofactor Diffie-Hellman primitive will likely involve consideration of security concerns like small subgroup attacks and the efficiency requirements of the application. Selection of parameter and public key validation methods will likely involve consideration of security issues like those discussed above. Selection of appropriate information to include in $SharedInfo_1$ and $SharedInfo_2$ will likely depend on the particular application, but common things to include are the public key R for alignment with the description of ECIES in [ABR01b], or a counter value to mitigate against replay of ciphertexts.

Additional information on ECIES, including extensive security discussion, can be found in ANS X9.63 [X9.63], the paper of Abdalla, Bellare, and Rogaway [ABR01b], and the book chapter of Dent

[Den05]. Test vectors for ECIES can be found in GEC 2 [GEC 2].

B.4.2 Commentary on Wrapped Key Transport Scheme

The wrapped key transport scheme is based on a NIST Special Publication [800-56A], which derives the scheme from parts of some IETF RFCs such as [2630, 3278].

B.5 Commentary on Section 6 — Key Agreement Schemes

This section provides commentary on Section 6 of the main body of this document.

B.5.1 Commentary on the Elliptic Curve Diffie-Hellman Scheme

The elliptic curve Diffie-Hellman scheme is a key agreement scheme based on ECC. It is designed to provide a variety of security goals depending on its application — goals it can provide include unilateral implicit key authentication, mutual implicit key authentication, known-key security, and forward secrecy. It is the elliptic curve analog of the Diffie-Hellman scheme [DH76]. It was first proposed in [DH76, Kob87, Mil85].

The elliptic curve Diffie-Hellman scheme was chosen for inclusion in this document because it is well-known, well-scrutinized, widely-standardized, and versatile. It is standardized in ANS X9.63 [X9.63], IEEE 1363 [1363], and ISO 15946-3 [15946-3]. Examples of the application of the elliptic curve Diffie-Hellman scheme to achieve a variety of security goals can be found in [BCK98, BWJM97, DvOW92]. ECIES is also an example of an application of the elliptic curve Diffie-Hellman scheme.

There are a number of known attack methods on the elliptic curve Diffie-Hellman scheme. The specification of the elliptic curve Diffie-Hellman scheme in this document includes provisions for preventing all these attacks. Nonetheless implementers should be aware of the attacks and monitor future advances. The attacks illustrate the importance of ECDH implementations performing all the security checks specified in the main body of this document. The following is a list of some of the known attacks:

- Attacks on the ECDLP or ECDHP. The security of the elliptic curve Diffie-Hellman scheme relies on the difficulty of the ECDLP and ECDHP on the elliptic curve domain parameters used — otherwise an attacker who sees a public key transmitted from U to V using the scheme may be able to recover the shared secret value z from Q_U and Q_V, and use this information to discover the keying data they agreed.

- Attacks on key generation. Key generation is involved in the key deployment procedure of the elliptic curve Diffie-Hellman scheme. Secure random or pseudorandom number generation is required during key generation to prevent, for example, V selecting a predictable private key. Insecure random and pseudorandom number generators are perhaps the most common cause of cryptographic attacks on cryptographic systems.

- Man-in-the-middle attacks. If the elliptic curve Diffie-Hellman scheme is not applied with care, it may be possible for an adversary to attack the scheme by modifying Q_U or Q_V when they are exchanged, and as a result prevent the scheme from achieving goals like implicit key authentication or known-key security. Numerous defenses are commonly employed to prevent such active attacks — including exchanging Q_U and Q_V in signed messages, or certifying Q_U and Q_V.

- Attacks on the key derivation function. The key derivation function used by the elliptic curve Diffie-Hellman scheme must possess a number of properties to ensure the security of the scheme. If, for example, an attacker is able to predict some bits of the output of the key derivation function, or if portions of the output of the key derivation function are correlated in some way, an attacker may be able to learn some information about the agreed keying data.

- Attacks based on the use of invalid domain parameters. The security of the elliptic curve Diffie-Hellman scheme relies on U and V using valid domain parameters because, for example, invalid domain parameters may be susceptible to the Pohlig-Hellman attack [PH78]. Entities U and V should therefore receive assurance that the elliptic curve domain parameters used are valid.

- Attacks based on the use of invalid public keys. Because the elliptic curve Diffie-Hellman scheme by its nature requires each entity to combine its private key with another entity's public key, the scheme is particularly susceptible to attacks based on the use of invalid public keys. The best-known examples of such attacks are small subgroup attacks [Joh96, LL97], which can result in, for example, an attacker coercing U and V into sharing predictable keying data, or an attacker learning some bits of U's private key. For this reason, when using the elliptic curve Diffie-Hellman scheme with the standard Diffie-Hellman primitive, U should receive an assurance that V's public key is valid and vice versa, and when using the scheme with the cofactor Diffie-Hellman primitive, U should receive an assurance that V's public key is partially valid and vice versa.

Of course a variety of non-cryptographic attacks on the elliptic curve Diffie-Hellman scheme are also possible, and implementers should take precautions to avoid, for example, "implementation attacks" such as fault-based attacks [BDL97], power-analysis attacks [KJJ99], and timing-analysis attacks [Koc96].

The operation of the elliptic curve Diffie-Hellman scheme involves selection by implementers of a number of options. These choices will typically be made based on concerns like efficiency, interoperability, and on security issues like those outlined above. In particular, some of the choices involved are selection of elliptic curve domain parameters, selection of a key derivation function, selection of the standard or cofactor Diffie-Hellman primitive, selection of parameter and public key validation methods, selection of *SharedInfo*, and selection of an appropriate application of the scheme to meet the security requirements of the system. Selection of elliptic curve domain parameters will likely involve consideration of issues like those discussed in Sections B.1 and B.2. Selection of the standard or cofactor Diffie-Hellman primitive will likely involve consideration of security concerns like small subgroup attacks and the efficiency requirements of the system. Selection of parameter

and public key validation methods will likely involve consideration of security issues like those discussed above. Selection of appropriate information to include in *SharedInfo* will likely depend on the particular application, but common things to include are the identities of U and V, the public keys Q_U and Q_V, counter values, and an indication of the symmetric scheme for which the agreed keying data will be used. If a number of fields are included in *SharedInfo*, it is sensible to check that the encoding of the fields is unique. Selection of an appropriate application of the scheme will likely depend on issues like what the agreed key will be used for and whether U and V are both on-line. ANS X9.63 contains guidance to help implementers make this selection.

Additional information on the elliptic curve Diffie-Hellman scheme, including extensive security discussion, can be found in ANS X9.63 [X9.63], and IEEE 1363 [1363]. Test vectors for the elliptic curve Diffie-Hellman scheme can be found in GEC 2 [GEC 2].

B.5.2 Commentary on the Elliptic Curve MQV Scheme

Like the elliptic curve Diffie-Hellman scheme, the elliptic curve MQV scheme is a key agreement scheme based on ECC. It is designed to provide a variety of security goals depending on its application — goals it can provide include mutual implicit key authentication, known-key security, and forward secrecy. It was first proposed in [LMQ+98, MQV95].

The elliptic curve MQV scheme was chosen for inclusion in this document because it is a particularly efficient method for achieving mutual implicit key authentication. The dominant calculations involved in the key agreement operation are 1.5 scalar multiplications. The elliptic curve MQV scheme is also standardized in ANS X9.63 [X9.63], and IEEE 1363 [1363].

There are a number of known attack methods on the elliptic curve MQV scheme. The specification of the elliptic curve MQV scheme in this document includes provision for preventing all these attacks. Nonetheless implementers should be aware of the attacks and monitor future advances. The attacks illustrate the importance of ECMQV implementations performing all the security checks specified in the main body of this document. The following is a list of some of the known attacks:

- Attacks on the ECDLP or ECDHP. The security of the elliptic curve MQV scheme relies on the difficulty of the ECDLP and ECDHP on the elliptic curve domain parameters used — otherwise an attacker who sees an U to V using the scheme may be able to recover the shared secret value z from $Q_{1,U}$, $Q_{2,U}$, $Q_{1,V}$, and $Q_{2,V}$, and use this information to discover the keying data they agreed.

- Attacks on key generation. Key generation is involved in the key deployment procedure of the elliptic curve MQV scheme. Secure random or pseudorandom number generation is required during key generation to prevent, for example, V selecting a predictable private key. Insecure random and pseudorandom number generators are perhaps the most common cause of cryptographic attacks on cryptographic systems.

- Attacks on the key derivation function. The key derivation function used by the elliptic curve MQV scheme must possess a number of properties to ensure the security of the scheme. If, for example, an attacker is able to predict some bits of the output of the key derivation

function, or if portions of the output of the key derivation function are correlated in some way, an attacker may be able to learn some information about the agreed keying data.

- Attacks based on the use of invalid domain parameters. The security of the elliptic curve MQV scheme relies on U and V using valid domain parameters because, for example, invalid domain parameters may be susceptible to the Pohlig-Hellman attack [PH78]. Entities U and V should therefore receive assurance that the elliptic curve domain parameters used are valid.
- Unknown key-share attacks. Kaliski [Kal98] has observed that the elliptic curve MQV scheme may be susceptible to unknown key-share attacks if it is not applied with care. These attacks may be damaging when the scheme is used to provide symmetric keys in order to both encrypt and authenticate data. The attacks can be prevented by including data like U and V's identities in *SharedInfo*, or by performing appropriate key confirmation subsequent to key agreement.

Of course a variety of non-cryptographic attacks on the elliptic curve MQV scheme are also possible, and implementers should take precautions to avoid, for example, "implementation attacks" such as fault-based attacks [BDL97], power-analysis attacks [KJJ99], and timing-analysis attacks [Koc96].

The operation of the elliptic curve MQV scheme involves selection by implementers of a number of options. These choices will typically be made based on concerns like efficiency, interoperability, and on security issues like those outlined above. In particular, some of the choices involved are selection of elliptic curve domain parameters, selection of a key derivation function, selection of parameter and public key validation methods, and selection of *SharedInfo*, as well as selection of an appropriate application of the scheme to meet the security requirements of the system. Selection of elliptic curve domain parameters will likely involve consideration of issues like those discussed in Section B.1 and B.2. Selection of parameter and public key validation methods will likely involve consideration of security issues like those discussed above. Selection of appropriate information to include in *SharedInfo* will likely depend on the particular application, but common things to include are the identities of U and V, the public keys $Q_{1,U}$, $Q_{2,U}$, $Q_{1,V}$, and $Q_{2,V}$, counter values, and an indication of the symmetric scheme for which the agreed keying data will be used. If a number of fields are included in *SharedInfo*, it is sensible to check that the encoding of the fields is unique. Selection of an appropriate application of the scheme will likely depend on issues like what the agreed key will be used for and whether U and V are both on-line. ANS X9.63 contains guidance to help implementers make this selection.

Additional information on the elliptic curve MQV scheme, including extensive security discussion, can be found in ANS X9.63 [X9.63], in IEEE 1363 [1363], and in the paper of Law, Menezes, Qu, Solinas, and Vanstone [LMQ$^+$98]. Test vectors for the elliptic curve MQV scheme can be found in GEC 2 [GEC 2].

B.6 Alignment with Other Standards

The cryptographic schemes in this document have been selected to conform with as many other standards efforts on ECC as possible. Standards efforts on ECC include:

- American National Standard Institute (ANSI) standards [X9.62b, X9.63],

- Institute of Electrical and Electronics Engineers (IEEE) standards [1363, 1363A],
- Internet Engineering Task Force (IETF) documents such as IPSec documents [2409, 4306, Int06a, 4753], TLS documents [2246, 4492], S/MIME documents [2630, 3278], and PKIX documents [3279, Int06b, 5480],
- International Standards Organization (ISO) standards [14888-3, 15946-1, 15946-2, 15946-3, 18033-2],
- New European Schemes for Signatures, Integrity and Encryption [NESSIE],
- National Institute for Standards and Technology (NIST) publications Federal Information Processing Standard (FIPS) [186-2], and Special Publications [800-56A] and [800-90].

Table 4 shows which of the schemes specified in this document are included in these efforts. More details are given below.

Standard	Schemes included
ANS X9.62	ECDSA
ANS X9.63	ECIES, ECDH, ECMQV
IEEE 1363	ECDSA, ECDH, ECMQV
IEEE 1363A	ECIES
IETF IPSec	ECDH, ECDSA
IETF TLS	ECDH, ECDSA
IETF PKIX	ECDSA
IETF S/MIME	ECDSA, ECDH, ECMQV, ECWKTS
IS 14888-3	ECDSA
IS 15946	ECDSA, ECDH, ECMQV
IS 18033-2	ECIES
NESSIE	ECDSA
NIST FIPS 186-2	ECDSA
NIST SP 800-56A	ECDH, ECMQV, ECWKTS
NIST SP 800-90	ECRNG

Table 4: Alignment with other ECC standards

The ANS X9.62 standard specifies ECDSA for use by the financial services industry. It requires ECDSA to be used with an ANSI-approved hash function, and with elliptic curve domain parameters with $n > 2^{160}$ to meet the stringent security requirements of the banking industry. Subject to these constraints and other procedural constraints such as the use of an ANSI-approved random

number generator, the specification of ECDSA in this document should comply with ANS X9.62-2005.

The draft ANS X9.63 standard specifies key agreement and key transport schemes based on ECC for use by the financial services industry. In particular it specifies various schemes built from ECIES, ECDH, and ECMQV. Like ANS X9.62, it requires various constraints such as restriction to an ANSI-approved hash function, use of elliptic curve domain parameters with $n > 2^{160}$, and use of an ANSI-approved pseudorandom number generator. Subject to these constraints, the specifications of ECDH and ECMQV in this document should be compatible with ANS X9.63. The specification of ECIES in this document should be similarly compatible with ANS X9.63 when used with the XOR symmetric encryption scheme. (ANS X9.63 is concerned specifically with key transport of short keys and hence support for a TDES or AES symmetric encryption option is not so desirable there as it is here where longer messages may be encrypted.)

The IEEE 1363 standard has a wide scope encompassing schemes based on the integer factorization problem, the traditional discrete logarithm problem, and the ECDLP. The techniques based on ECC specified in IEEE 1363 include general descriptions of ECDSA (known in IEEE 1363 as ECSSA), ECDH (known as ECKAS-DH1), and ECMQV (known as ECKAS-MQV). The specifications of ECDSA, ECDH, and ECMQV in this document should comply with IEEE 1363.

The IEEE 1363A standard is an addendum of IEEE 1363 and includes additional public key cryptography schemes. In particular, it includes ECIES.

The IPSec standards include support for a variant of ECDH. The particular variant of ECDH differs from ECDH as specified in this document in as much as it uses different octet string point representations. In addition, the default elliptic curve domain parameters in IPSec are parameters over $\mathbb{F}_{2^m}$ with m composite and they do not use prime order base points. Draft updates to IPSec will allow a greater variety of elliptic curves. Aside from these technicalities, the specification of ECDH in this document should be broadly compliant with IPSec. Also there is support for using ECDSA as an authentication mechanism in IPSec.

The TLS standards include support for ECDH and ECDSA that are broadly compliant with this document. One distinction to note (which also applies to IPSec) is that TLS uses its own key derivation function.

The S/MIME standards include support for ECDSA, ECDH and ECMQV that are strongly compliant to this standard.

The PKIX standards include support for ECDSA, specifically for certificate authorities to use for signing a certificate. They overlap considerably with Appendix C of this document. Indirect support for other algorithms is anticipated, primarily through indications in the certificate for which algorithm of ECDSA, ECDH, or ECMQV the public key is intended, but perhaps also for providing proof-of-possession.

The IS 14888-3 standard specifies very general signature mechanisms. The specification of ECDSA in this document should comply with IS 14888-3.

The IS 15946 standards specify cryptographic techniques based on ECC. IS 15946-2 specifies a variety of signature schemes including ECDSA. The specification of ECDSA in this document should comply with IS 15946-2. IS 15946-3 specifies a variety of key establishment schemes including some based on ECDH and ECMQV. The specifications of ECDH and ECMQV in this document should

be compatible with IS 15946-3.

The draft IS 18033 standards specify encryption techniques schemes. IS 18033-2 specifies a variety of asymmetric encryption techniques, including ECIES.

The NESSIE report recommends a variety of cryptographic techniques. The NESSIE report includes implementation specifics, security analysis, and performance analysis for each technique. One of the recommended signature schemes is ECDSA. The April 2004 draft [NESSIE] however, excludes a check that $r \neq 0$, which makes ECDSA insecure if the verifier does not know how the base point G was generated.

The NIST publication FIPS 186-2 specifies two digital signature algorithms, one being ECDSA. For the most part, the specification is by reference to ANS X9.62. A replacement FIPS 186-3 is underway, and will continue to include ECDSA. The specification of ECDSA here is compliant with the current draft of FIPS 186-3 [186-3].

The NIST Special Publication 800-56A specifies a variety of techniques for key establishment, including ECDH and ECMQV. The versions of ECDH and ECMQV specified here can be implemented in a way compliant to NIST SP 800-56A.

The NIST Special Publication 800-90 specifies a variety of deterministic random number generators, including the Dual_EC_DRBG, which is equivalent to ECRNG specified here. For certain choices of parameters, the ECRNG is compliant to NIST SP 800-90.

C ASN.1 for Elliptic Curve Cryptography

This section specifies the ASN.1 syntax that should be used when ASN.1 syntax is used to convey parts of this information. Generally, the ASN.1 syntax needs to be suitably encoded via, for example, DER [X.690]. Several different types of information may need to be conveyed during the operation of the schemes specified in this document. Section C.1 recommends syntax to describe finite fields. Section C.2 recommends syntax to describe elliptic curve domain parameters. Section C.3 recommends syntax to describe elliptic curve public keys. Section C.4 recommends syntax to describe elliptic curve private keys. Section C.5 recommends syntax to describe signature and key establishment schemes. Section C.6 recommends syntax to encode information for processing key derivation functions. Section C.7 recommends syntax to encode a protocol data unit, if this is needed. Section C.8 contains the ASN.1 module that holds all the syntax above.

Syntax for other aspects of elliptic curve cryptography, such as object identifiers for specific schemes, may be added in future versions of this document. The syntax provided here profiles the syntax used in ANS X9.62 [X9.62b] and PKIX [3279, Int06b, 5480].

C.1 Syntax for Finite Fields

This section provides the recommended ASN.1 syntax to identify finite fields and specific elements of said fields. The identity of a finite field and a specific field element therein may need to be specified, for example, as part of some elliptic curve domain parameters. The syntax follows ANS X9.62 [X9.62b].

The finite fields of interest in this document are prime fields and characteristic-two fields. A finite field is identified by a value of type `FieldID`:

```
FieldID { FIELD-ID:IOSet } ::= SEQUENCE { -- Finite field
      fieldType FIELD-ID.&id({IOSet}),
      parameters FIELD-ID.&Type({IOSet}{@fieldType})
}
```

The governor `FIELD-ID` of the parameter `FIELD-ID:IOSet` is defined as the following open type that is defined in ITU-T X.681 [X.681, Annex A].

```
FIELD-ID ::= TYPE-IDENTIFIER
```

(and hence the dummy argument `IOSet` must be of said type). The term "open type" means that a "hole" is left in both components that are filled at the time of usage. The component relation constraint `{IOSet}{@fieldType}` binds the argument `IOSet` to the identifier `fieldType`.

Only two field types are permitted, namely, prime fields and characteristic-two fields.

```
FieldTypes FIELD-ID ::= {
      { Prime-p IDENTIFIED BY prime-field } |
      { Characteristic-two IDENTIFIED BY characteristic-two-field }
}
```

A prime field is specified by the identifier `prime-field` of type `Prime-p` (below) comprising an integer which is the size of the field.

```
prime-field OBJECT IDENTIFIER ::= { id-fieldType 1 }
Prime-p ::= INTEGER -- Field of size p.
```

The object identifier `id-fieldType` (above) is the root of a tree containing object identifiers of each field type. It is defined as the following arc from ANSI.

```
id-fieldType OBJECT IDENTIFIER ::= { ansi-X9-62 fieldType(1)}
```

where

```
ansi-X9-62 OBJECT IDENTIFIER ::= {
      iso(1) member-body(2) us(840) 10045
}
```

A characteristic-two finite field is specified by the identifier `characteristic-two-field` of type `Characteristic-two` (below) comprising the size of the field, the type of basis used to express elements of the field, and the polynomial used to generate the field (in the case of a polynomial basis).

```
characteristic-two-field OBJECT IDENTIFIER ::= { id-fieldType 2 }
Characteristic-two ::= SEQUENCE {
      m INTEGER, -- Field size 2^m
      basis CHARACTERISTIC-TWO.&id({BasisTypes}),
      parameters CHARACTERISTIC-TWO.&Type({BasisTypes}{@basis})
}
```

The type `CHARACTERISTIC-TWO` (above) is defined by the following.

```
CHARACTERISTIC-TWO ::= TYPE-IDENTIFIER
```

The basis types of interest are normal bases (that are not used here), trinomial bases, or pentanomial bases. (See Section 2.1.2 for further information.)

```
BasisTypes CHARACTERISTIC-TWO ::= {
      { NULL IDENTIFIED BY gnBasis } |
      { Trinomial IDENTIFIED BY tpBasis } |
      { Pentanomial IDENTIFIED BY ppBasis },
      ...
}
```

Normal bases are specified by the object identifier `gnBasis` (below) with `NULL` parameters. Trinomial bases are specified by the object identifier `tpBasis` (below) with a parameter `Trinomial` that

specifies the degree of the middle term in the defining trinomial. Pentanomial bases are specified by the object identifier `ppBasis` (below) with a parameter `Pentanomial` that specifies the degrees of the three middle terms in the defining pentanomial.

```
gnBasis OBJECT IDENTIFIER ::= { id-characteristic-two-basis 1 }
tpBasis OBJECT IDENTIFIER ::= { id-characteristic-two-basis 2 }
ppBasis OBJECT IDENTIFIER ::= { id-characteristic-two-basis 3 }
```

The identifier `id-characteristic-two-basis` (above) is defined as the following.

```
id-characteristic-two-basis OBJECT IDENTIFIER ::= {
      characteristic-two-field basisType(3)
}
```

The degrees of the polynomials that define the finite fields are specified by the following.

```
Trinomial ::= INTEGER
Pentanomial ::= SEQUENCE {
      k1 INTEGER, -- k1 > 0
      k2 INTEGER, -- k2 > k1
      k3 INTEGER -- k3 > k2
}
```

Finally, a specific field element is represented by the following type

```
FieldElement ::= OCTET STRING
```

whose value is the octet string obtained from the conversion routines given in Section 2.3.5.

C.2 Syntax for Elliptic Curve Domain Parameters

Elliptic curve domain parameters may need to be specified, for example, during the setup operation of a cryptographic scheme based on elliptic curve cryptography. There are a number of ways of specifying elliptic curve domain parameters. Here the following syntax, as a choice of three parameters, is recommended (following [3279, Int06b, 5480]) for use in X.509 certificates and elsewhere.

```
ECDomainParameters{ECDOMAIN:IOSet} ::= CHOICE {
      specified SpecifiedECDomain,
      named ECDOMAIN.&id({IOSet}),
      implicitCA NULL
}
```

The choice of three parameters representation methods (above) allows detailed specification of all required values using either:

- The choice `specified` which explicitly identifies all the parameters, or,
- The choice `named` as an object identifier identifying a particular set of elliptic curve domain parameters, or
- The choice `implicitCA` which indicates that the parameters are the same as those of certification authority certifying the public key.

The valid values for the `namedCurve` choice are constrained to those within the class `ECDOMAIN` (defined below and explained further in SEC 2 [SEC 2]).

The following syntax is used to describe explicit representations of elliptic curve domain parameters, if need be. The inclusion of the cofactor is strongly recommended.

```
SpecifiedECDomain ::= SEQUENCE {
      version SpecifiedECDomainVersion(ecdpVer1 | ecdpVer2 | ecdpVer3, ...),
      fieldID FieldID {{FieldTypes}},
      curve Curve,
      base ECPoint,
      order INTEGER,
      cofactor INTEGER OPTIONAL,
      hash HashAlgorithm OPTIONAL,
      ...
}
```

The components of type `SpecifiedECDomain` have the following meanings.

- The component `version` is the version number of the ASN.1 type with a value of `1`, `2` or `3`. The notation above is used to constrain `version` to a set of values. The meaning of these three values are as follows. If `version` is `ecdpVer2`, then the curve and the base point shall be generated verifiably at random, and `curve.seed` shall be present. If `version` is `ecdpVer3`, then the curve is not generated verifiably at random but the base point is, and `curve.seed` shall be present.
- The component `fieldID` identifies the finite field over which the elliptic curve is defined and was discussed in Section C.1.
- The component `curve` of type `Curve` (defined below) specifies the elliptic curve.
- The component `base` of type `ECPoint` (defined below) specifies the base point on the elliptic curve `curve`.
- The component `order` is the order of the base point `base`.
- The component `cofactor` is the order of the curve divided by the order of the base point. Inclusion of the cofactor is optional – however, it is recommended that that the cofactor be included in order to facilitate interoperability between implementations.

- The component `hash` is the hash function used to generate the domain parameters verifiably at random.

The type `SpecifiedECDomainVersion` is a subtype of `INTEGER` and is used to constrain the set of versions.

```
SpecifiedECDomainVersion ::= INTEGER {
      ecdpVer1(1),
      ecdpVer2(2),
      ecdpVer3(3)
}
```

The type `Curve` is defined as follows, by specifying the coefficients of the defining equation of the elliptic curve and an optional seed. (If the curve was generated verifiably at random using a seed value with a hash function such as SHA-1 as specified in ANS X9.62 [X9.62b] then said seed may be included as the `seed` component so as to allow a recipient to verify that the curve was indeed so generated using said seed.)

```
Curve ::= SEQUENCE {
      a FieldElement,
      b FieldElement,
      seed BIT STRING OPTIONAL
      -- Shall be present if used in SpecifiedECDomain
      -- with version equal to ecdpVer2 or ecdpVer3
}
```

An elliptic curve point itself is represented by the following type

```
ECPoint ::= OCTET STRING
```

whose value is the octet string obtained from the conversion routines given in Section 2.3.3.

The class `ECDOMAIN`, defined as follows, is used to specify a named curve.

```
ECDOMAIN ::= CLASS {
      &id OBJECT IDENTIFIER UNIQUE
}
WITH SYNTAX { ID &id }
```

For example, the curve `sect163k1`, defined in SEC 2 [SEC 2], is denoted by the syntax `ID sect163k1`.

The following syntax, included here for completeness, may be extended by other standards and implementations to specify the list of supported named curves. One such extension may be found in SEC 2 [SEC 2] ; another such extension may be found in ANS X9.62 [X9.62b].

```
SECGCurveNames ECDOMAIN::= {
      ...  -- named curves
```

```
}
```

The following type `HashAlgorithm` is used to specify a hash function.

```
HashAlgorithm ::= AlgorithmIdentifier {{ HashFunctions }}
```

The information object set `HashFunctions` specifies the allowed hash functions currently:

```
HashFunctions ALGORITHM ::= {
      {OID sha-1 PARMS NULL } |
      {OID id-sha224 PARMS NULL } |
      {OID id-sha256 PARMS NULL } |
      {OID id-sha384 PARMS NULL } |
      {OID id-sha512 PARMS NULL } ,
      ...  -- Additional hash functions may be added in the future }
```

When the parameters field of `HashAlgorithm` is constrained to the type `NULL`, then the parameters should be omitted.

The following object identifiers are used above to identify specific hash functions.

```
sha-1 OBJECT IDENTIFIER ::= {iso(1) identified-organization(3)
      oiw(14) secsig(3) algorithm(2) 26}
id-sha OBJECT IDENTIFIER ::= { joint-iso-itu-t(2) country(16) us(840)
      organization(1) gov(101) csor(3) nistalgorithm(4) hashalgs(2) }
id-sha224 OBJECT IDENTIFIER ::= { id-sha 4 }
id-sha256 OBJECT IDENTIFIER ::= { id-sha 1 }
id-sha384 OBJECT IDENTIFIER ::= { id-sha 2 }
id-sha512 OBJECT IDENTIFIER ::= { id-sha 3 }
```

C.3 Syntax for Elliptic Curve Public Keys

Elliptic curve public keys may need to be specified, for example, during the key deployment phase of a cryptographic scheme based on elliptic curve cryptography. An elliptic curve public key is a point on an elliptic curve and may be represented in a variety of ways using ASN.1 syntax. Here the following syntax is recommended (following [3279, Int06b, 5480]) for use in X.509 certificates and elsewhere, where public keys are represented by the ASN.1 type `SubjectPublicKeyInfo`.

```
SubjectPublicKeyInfo ::= SEQUENCE {
      algorithm AlgorithmIdentifier {{ECPKAlgorithms}} (WITH COMPONENTS
            {algorithm, parameters}) ,
      subjectPublicKey BIT STRING
}
```

The component `algorithm` specifies the type of public key and associated parameters employed

and the component `subjectPublicKey` specifies the actual value of said public key.

The parameter type `AlgorithmIdentifier` above tightly binds together a set of algorithm object identifiers and their associated parameters types. The type `AlgorithmIdentifier` is defined as follows.

```
AlgorithmIdentifier{ ALGORITHM:IOSet } ::= SEQUENCE {
      algorithm ALGORITHM.&id({IOSet}),
      parameters ALGORITHM.&Type({IOSet}{@algorithm}) OPTIONAL
}
```

The governing type `ALGORITHM` (above) is defined to be the following object information object.

```
ALGORITHM ::= CLASS {
      &id OBJECT IDENTIFIER UNIQUE,
      &Type OPTIONAL
}
      WITH SYNTAX { OID &id [PARMS &Type] }


ECPKAlgorithms ALGORITHM ::= {
      ecPublicKeyType |
      ecPublicKeyTypeRestricted |
      ecPublicKeyTypeSupplemented |
      {OID ecdh PARMS ECDomainParameters {{SECGCurveNames}}} |
      {OID ecmqv PARMS ECDomainParameters {{SECGCurveNames}}},
      ...
}
ecPublicKeyType ALGORITHM ::= {
      OID id-ecPublicKey PARMS ECDomainParameters {{SECGCurveNames}}
}
```

The object identifier `id-ecPublicKey` designates an elliptic curve public key. It is defined by the following (after ANS X9.62 [X9.62b]) to be used whenever an object identifier for an elliptic curve public key is needed. (Note that this syntax applies to all elliptic curve public keys regardless of their designated use.)

```
id-ecPublicKey OBJECT IDENTIFIER ::= { id-publicKeyType 1 }
```

where

```
id-publicKeyType OBJECT IDENTIFIER ::= { ansi-X9-62 keyType(2) }
```

The following information object of class `ALGORITHM` indicates the type of the parameters component of an `AlgorithmIdentifier {}` containing the OID `id-ecPublicKeyRestricted`.

```
ecPublicKeyTypeRestricted ALGORITHM ::= {
      OID id-ecPublicKeyTypeRestricted PARMS ECPKRestrictions
```

```
}
```

The OID `id-ecPublicKeyTypeRestricted` is used to identify a public key that has restrictions on which ECC algorithms it can be used with.

```
id-ecPublicKeyTypeRestricted OBJECT IDENTIFIER ::= {
      id-publicKeyType restricted(2) }
```

The type `ECPKRestrictions` identifies the restrictions on the algorithms that can be used with a given elliptic curve public key.

```
ECPKRestrictions ::= SEQUENCE {
      ecDomain ECDomainParameters {{ SECGCurveNames }},
      eccAlgorithms ECCAlgorithms
}
```

The type `ECCAlgorithms` is used to identify one or more ECC algorithms, possibly, but not necessarily, in an order of preference.

```
ECCAlgorithms ::= SEQUENCE OF ECCAlgorithm
```

The type `ECCAlgorithm` is a constrained instance of the parameterized type `AlgorithmIdentifier {}`, and is used to identify an ECC algorithm.

```
ECCAlgorithm ::= AlgorithmIdentifier {{ECCAlgorithmSet}}
```

When the optional parameters field of `ECCAlgorithm` is constrained to the type `NULL`, then it should be omitted. When the optional parameters field is constrained to a type other than `NULL`, then it should be present.

The component `ECDomainParameters` was defined in Section C.2 and may contain the elliptic curve domain parameters associated with the public key in question. (Thus the component `algorithm` indicates that `SubjectPublicKeyInfo` not only specifies the elliptic curve public key but also the elliptic curve domain parameters associated with said public key.)

Finally, `SubjectPublicKeyInfo` specifies the public key itself when `algorithm` indicates that the public key is an elliptic curve public key.

The elliptic curve public key (a value of type `ECPoint` that is an `OCTET STRING`) is mapped to a `subjectPublicKey` (a value encoded as type `BIT STRING`) as follows: The most significant bit of the value of the `OCTET STRING` becomes the most significant bit of the value of the `BIT STRING` and so on with consecutive bits until the least significant bit of the `OCTET STRING` becomes the least significant bit of the `BIT STRING`.

The following information object of class `ALGORITHM` indicates the type of the paramaters component of an `AlgorithmIdentifier {}` containing the OID `id-ecPublicKeySupplemented`.

```
ecPublicKeyTypeSupplemented ALGORITHM ::= {
      OID id-ecPublicKeyTypeSupplemented PARMS ECPKSupplements
```

```
}
```

The OID id-ecPublicKeyTypeSupplemented is used to identify a public key that has restrictions on which ECC algorithms it can be used with.

```
secg-scheme OBJECT IDENTIFIER ::= { iso(1)
      identified-organization(3) certicom(132) schemes(1) }
id-ecPublicKeyTypeSupplemented OBJECT IDENTIFIER ::= {
      secg-scheme supplementalPoints(0) }
```

The type ECPKSupplements identifies the supplements (and restrictions) on the algorithms that can be used with a given elliptic curve public key.

```
ECPKSupplements ::= SEQUENCE {
      ecDomain ECDomainParameters {{ SECGCurveNames }},
      eccAlgorithms ECCAlgorithms,
      eccSupplements ECCSupplements }
```

The type ECCSupplements serves to provide a list of multiples of the public key. These multiples can be used to accelerate the public key operations necessary with that public key.

```
ECCSupplements ::= CHOICE {
      namedMultiples [0] NamedMultiples,
      specifiedMultiples [1] SpecifiedMultiples
}
NamedMultiples ::= SEQUENCE {
      multiples OBJECT IDENTIFIER,
      points SEQUENCE OF ECPoint }
SpecifiedMultiples ::= SEQUENCE OF SEQUENCE {
      multiple INTEGER,
      point ECPoint }
```

C.4 Syntax for Elliptic Curve Private Keys

An elliptic curve private key may need to be conveyed, for example, during the key deployment operation of a cryptographic scheme in which a Certification Authority generates and distributes the private keys. An elliptic curve private key is an unsigned integer. The following ASN.1 syntax may be used.

```
ECPrivateKey ::= SEQUENCE {
      version INTEGER { ecPrivkeyVer1(1) } (ecPrivkeyVer1),
      privateKey OCTET STRING,
      parameters [0] ECDomainParameters {{ SECGCurveNames }} OPTIONAL,
      publicKey [1] BIT STRING OPTIONAL
}
```

where

- The component `version` specifies the version number of the elliptic curve private key structure. The syntax above creates the element `ecPrivkeyVer1` of type `INTEGER` whose value is `1`.
- The component `privateKey` is the private key defined to be the octet string of length $\lceil \log_2 n/8 \rceil$ (where n is the order of the curve) obtained from the unsigned integer via the encoding of Section 2.3.7.
- The optional component `parameters` specifies the elliptic curve domain parameters associated to the private key. The type `Parameters` was discussed in Section C.2. If the parameters are known by other means then this component may be `NULL` or omitted.
- The optional component `publicKey` contains the elliptic curve public key associated with the private key in question. Public keys were discussed in Section C.3. It may be useful to send the public key along with the private key, especially in a scheme such as MQV that involves calculations with the public key.

The syntax for `ECPrivateKey` may be used, for example, to convey elliptic curve private keys using the syntax for `PrivateKeyInfo` as defined in PKCS #8 [PKCS8]. In such a case, the value of the component `privateKeyAlgorithm` within `PrivateKeyInfo` shall be `id-ecPublicKey` as discussed in Section C.3 above.

C.5 Syntax for Signature and Key Establishment Schemes

Signatures may need to be conveyed from one party to another whenever ECDSA is used to sign a message. The following syntax is recommended to represent actual signatures for use within X.509 certificates, CRLs (following [3279, Int06b, 5480]), and elsewhere. The signature is conveyed using the parameterized type `SIGNED`. It comprises the specification of an algorithm of type `AlgorithmIdentifier` together with the actual signature

When the signature is generated using ECDSA with SHA-1, the algorithm component shall contain the object identifier `ecdsa-with-SHA1` (defined below) and the parameters component shall either contain `NULL` or be absent. The parameters component should be omitted.

```
ecdsa-with-SHA1 OBJECT IDENTIFIER ::= { id-ecSigType sha1(1)}
ecdsa-with-Recommended OBJECT IDENTIFIER ::= { id-ecSigType recommended(2) }
ecdsa-with-Specified OBJECT IDENTIFIER ::= { id-ecSigType specified(3)}
ecdsa-with-Sha224 OBJECT IDENTIFIER ::= { id-ecSigType specified(3) 1 }
ecdsa-with-Sha256 OBJECT IDENTIFIER ::= { id-ecSigType specified(3) 2 }
ecdsa-with-Sha384 OBJECT IDENTIFIER ::= { id-ecSigType specified(3) 3 }
ecdsa-with-Sha512 OBJECT IDENTIFIER ::= { id-ecSigType specified(3) 4 }
id-ecSigType OBJECT IDENTIFIER ::= { ansi-X9-62 signatures(4) }
```

The information object set `ECDSAAlgorithmSet` specifies how the object identifiers above are to be used in algorithm identifiers and also serves to constrain the set of algorithms specifiable in this ASN.1 syntax, when using ECDSA.

```
ECDSAAlgorithmSet ALGORITHM ::= {
      {OID ecdsa-with-SHA1 PARMS NULL} |
      {OID ecdsa-with-Recommended PARMS NULL} |
      {OID ecdsa-with-Specified PARMS HashAlgorithm } |
      {OID ecdsa-with-Sha224 PARMS NULL} |
      {OID ecdsa-with-Sha256 PARMS NULL} |
      {OID ecdsa-with-Sha384 PARMS NULL} |
      {OID ecdsa-with-Sha512 PARMS NULL} ,
      ...  -- More algorithms need to be added
}
```

The information object set `ECCAlgorithmSet` specifies the ECC algorithms that can be identified with this syntax.

```
ECCAlgorithmSet ALGORITHM ::= {
      ECDSAAlgorithmSet |
      ECDHAlgorithmSet |
      ECMQVAlgorithmSet |
      ECIESAlgorithmSet |
      ECWKTAlgorithmSet ,
      ...
}
```

The information object set `ECDHAlgorithmSet` used above is defined below.

```
ECDHAlgorithmSet ALGORITHM ::= {
      {OID dhSinglePass-stdDH-sha1kdf PARMS NULL} |
      {OID dhSinglePass-cofactorDH-sha1kdf PARMS NULL} |
      {OID dhSinglePass-cofactorDH-recommendedKDF} |
      {OID dhSinglePass-cofactorDH-specifiedKDF PARMS KeyDerivationFunction} |
      {OID ecdh} |
      {OID dhSinglePass-stdDH-sha256kdf-scheme} |
      {OID dhSinglePass-stdDH-sha384kdf-scheme} |
      {OID dhSinglePass-stdDH-sha224kdf-scheme} |
      {OID dhSinglePass-stdDH-sha512kdf-scheme} |
      {OID dhSinglePass-cofactorDH-sha256kdf-scheme} |
      {OID dhSinglePass-cofactorDH-sha384kdf-scheme} |
      {OID dhSinglePass-cofactorDH-sha224kdf-scheme} |
      {OID dhSinglePass-cofactorDH-sha512kdf-scheme} ,
      ...  -- Future combinations may be added
}
```

The information object set ECMQVHAlgorithmSet used above is defined below.

```
ECMQVAlgorithmSet ALGORITHM ::= {
      {OID mqvSinglePass-sha1kdf} |
      {OID mqvSinglePass-recommendedKDF} |
      {OID mqvSinglePass-specifiedKDF PARMS KeyDerivationFunction} |
      {OID mqvFull-sha1kdf} |
      {OID mqvFull-recommendedKDF} |
      {OID mqvFull-specifiedKDF PARMS KeyDerivationFunction} |
      {OID ecmqv} |
      {OID mqvSinglePass-sha256kdf-scheme } |
      {OID mqvSinglePass-sha384kdf-scheme } |
      {OID mqvSinglePass-sha224kdf-scheme } |
      {OID mqvSinglePass-sha512kdf-scheme } |
      {OID mqvFull-sha256kdf-scheme } |
      {OID mqvFull-sha384kdf-scheme } |
      {OID mqvFull-sha224kdf-scheme } |
      {OID mqvFull-sha512kdf-scheme } ,
      ...  -- Future combinations may be added
}
```

The object identifiers used in the two information object sets above are given below.

```
x9-63-scheme OBJECT IDENTIFIER ::= { iso(1) member-body(2)
      us(840) ansi-x9-63(63) schemes(0) }
dhSinglePass-stdDH-sha1kdf OBJECT IDENTIFIER ::= {x9-63-scheme 2}
dhSinglePass-cofactorDH-sha1kdf OBJECT IDENTIFIER ::= {x9-63-scheme 3}
mqvSinglePass-sha1kdf OBJECT IDENTIFIER ::= {x9-63-scheme 16}
mqvFull-sha1kdf OBJECT IDENTIFIER ::= {x9-63-scheme 17}
dhSinglePass-cofactorDH-recommendedKDF OBJECT IDENTIFIER ::= {secg-scheme 1}
dhSinglePass-cofactorDH-specifiedKDF OBJECT IDENTIFIER ::= {secg-scheme 2}
ecdh OBJECT IDENTIFIER ::= {secg-scheme 12}
dhSinglePass-stdDH-sha256kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 11 1}
dhSinglePass-stdDH-sha384kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 11 2}
dhSinglePass-stdDH-sha224kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 11 0}
dhSinglePass-stdDH-sha512kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 11 3}
dhSinglePass-cofactorDH-sha256kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 14 1}
dhSinglePass-cofactorDH-sha384kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 14 2}
dhSinglePass-cofactorDH-sha224kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 14 0}
dhSinglePass-cofactorDH-sha512kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 14 3}
mqvSinglePass-recommendedKDF OBJECT IDENTIFIER ::= {secg-scheme 3}
mqvSinglePass-specifiedKDF OBJECT IDENTIFIER ::= {secg-scheme 4}
mqvFull-recommendedKDF OBJECT IDENTIFIER ::= {secg-scheme 5}
mqvFull-specifiedKDF OBJECT IDENTIFIER ::= {secg-scheme 6}
ecmqv OBJECT IDENTIFIER ::= {secg-scheme 13}
mqvSinglePass-sha256kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 15 1}
```

```
mqvSinglePass-sha384kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 15 2}
mqvSinglePass-sha224kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 15 0}
mqvSinglePass-sha512kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 15 3}
mqvFull-sha256kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 16 1}
mqvFull-sha384kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 16 2}
mqvFull-sha224kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 16 0}
mqvFull-sha512kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 16 3}
```

The object identifiers above that end in `recommendedKDF` indicated that key derivation to use is the default for the associated elliptic curve domain parameters. The object identifiers `ecdh` and `ecmqv` are meant for very general indication, with other details to be specified out of band.

The type `KeyDerivationFunction` is given below.

```
KeyDerivationFunction ::= AlgorithmIdentifier {{KDFSet}}
KDFSet ALGORITHM ::= {
      {OID x9-63-kdf PARMS HashAlgorithm } |
      {OID nist-concatenation-kdf PARMS HashAlgorithm } |
      {OID tls-kdf PARMS HashAlgorithm } |
      {OID ikev2-kdf PARMS HashAlgorithm } ,
      ...  -- Future combinations may be added
}
x9-63-kdf OBJECT IDENTIFIER ::= {secg-scheme 17 0}
nist-concatenation-kdf OBJECT IDENTIFIER ::= {secg-scheme 17 1}
tls-kdf OBJECT IDENTIFIER ::= {secg-scheme 17 2}
ikev2-kdf OBJECT IDENTIFIER ::= {secg-scheme 17 3}
```

The information object set `ECIESAlgorithmSet` specifies how one identifies ECIES.

```
ECIESAlgorithmSet ALGORITHM ::= {
      {OID ecies-recommendedParameters} |
      {OID ecies-specifiedParameters PARMS ECIESParameters} ,
      ...  -- Future combinations may be added
}
```

The object identifiers given above are:

```
ecies-recommendedParameters OBJECT IDENTIFIER ::= {secg-scheme 7}
ecies-specifiedParameters OBJECT IDENTIFIER ::= {secg-scheme 8}
```

The type `ECIESParameters` is defined below.

```
ECIESParameters ::= SEQUENCE {
      kdf [0] KeyDerivationFunction OPTIONAL,
      sym [1] SymmetricEncryption OPTIONAL,
      mac [2] MessageAuthenticationCode OPTIONAL
}
```

```
SymmetricEncryption ::= AlgorithmIdentifier {{SYMENCSet}}
MessageAuthenticationCode ::= AlgorithmIdentifier {{MACSet}}
SYMENCSet ALGORITHM ::= {
      { OID xor-in-ecies } |
      { OID tdes-cbc-in-ecies } |
      { OID aes128-cbc-in-ecies } |
      { OID aes192-cbc-in-ecies } |
      { OID aes256-cbc-in-ecies } |
      { OID aes128-ctr-in-ecies } |
      { OID aes192-ctr-in-ecies } |
      { OID aes256-ctr-in-ecies } ,
      ...  -- Future combinations may be added
}
MACSet ALGORITHM ::= {
      { OID hmac-full-ecies PARMS HashAlgorithm} |
      { OID hmac-half-ecies PARMS HashAlgorithm} |
      { OID cmac-aes128-ecies } |
      { OID cmac-aes192-ecies } |
      { OID cmac-aes256-ecies } ,
      ...  -- Future combinations may be added
}
xor-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 18 }
tdes-cbc-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 19 }
aes128-cbc-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 20 0 }
aes192-cbc-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 20 1 }
aes256-cbc-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 20 2 }
aes128-ctr-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 21 0 }
aes192-ctr-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 21 1 }
aes256-ctr-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 21 2 }
hmac-full-ecies OBJECT IDENTIFIER ::= {secg-scheme 22 }
hmac-half-ecies OBJECT IDENTIFIER ::= {secg-scheme 23 }
cmac-aes128-ecies OBJECT IDENTIFIER ::= {secg-scheme 24 0 }
cmac-aes192-ecies OBJECT IDENTIFIER ::= {secg-scheme 24 1 }
cmac-aes256-ecies OBJECT IDENTIFIER ::= {secg-scheme 24 2 }
```

The information object set `ECWKTAlgorithmSet` specifies how one identifies elliptic curve wrapped key transport, if one is using the scheme as a single unit, not as a combination of the key agreement scheme and key wrap scheme. Typically, one may identify a wrapped key transport scheme separately as a combination of a key agreement schemes and key wrap scheme.

```
ECWKTAlgorithmSet ALGORITHM ::= {
      {OID ecwkt-recommendedParameters} |
      {OID ecwkt-specifiedParameters PARMS ECWKTParameters} ,
      ...  -- Future combinations may be added
}
```

The object identifiers given above are:

```
ecwkt-recommendedParameters OBJECT IDENTIFIER ::= {secg-scheme 9}
ecwkt-specifiedParameters OBJECT IDENTIFIER ::= {secg-scheme 10}
```

The type `ECWKTParameters` are defined below.

```
ECWKTParameters ::= SEQUENCE {
      kdf [0] KeyDerivationFunction OPTIONAL,
      wrap [1] KeyWrapFunction OPTIONAL
}
KeyWrapFunction ::= AlgorithmIdentifier {{KeyWrapSet}}
KeyWrapSet ALGORITHM ::= {
      { OID aes128-key-wrap } |
      { OID aes192-key-wrap } |
      { OID aes256-key-wrap } ,
      ...  -- Future combinations may be added
}
aes128-key-wrap OBJECT IDENTIFIER ::= {secg-scheme 25 0 }
aes192-key-wrap OBJECT IDENTIFIER ::= {secg-scheme 25 1 }
aes256-key-wrap OBJECT IDENTIFIER ::= {secg-scheme 25 2 }
```

The actual value of an ECDSA signature, that is, a signature identified by `ecdsa-with-SHA1` or any other of the above identifiers for ECDSA, is encoded as follows.

```
ECDSA-Signature ::= CHOICE {
      two-ints-plus ECDSA-Sig-Value,
      point-int [0] ECDSA-Full-R,
      ...  -- Future representations may be added
}
```

Note the first choice is a type compatible with the previous version of this standard. The second choice is an alternative format, which aims to provide a simpler means to aid accelerated methods of ECDSA verification. Because both choice alternative syntaxes are sequences and the rules of ASN.1 dictate that choices have different tags, the second choice has been tagged. The first choice is not tagged so that old signature will appear to comply.

The original syntax `ECDSA-Sig-Value` has been extended to allow for additional information to be attached which the verifier can use recover the value of R from r, permitting accelerated signature verification.

```
ECDSA-Sig-Value ::= SEQUENCE {
      r INTEGER,
      s INTEGER,
      a INTEGER OPTIONAL,
      y CHOICE { b BOOLEAN, f FieldElement } OPTIONAL
```

```
}
```

The alternative syntax for identifying an ECDSA signature value explicit includes the point R represented as an octet string.

```
ECDSA-Full-R ::= SEQUENCE {
      r ECPoint,
      s INTEGER
}
```

X.509 certificates and CRLs represent a signature as a bit string; in such cases, the entire encoding of a value of `ECDSA-Signature` is the value of said bit string.

The actual value of an ECIES ciphertext may be encoded in ASN.1 with the following type.

```
ECIES-Ciphertext-Value ::= SEQUENCE {
      ephemeralPublicKey ECPoint,
      symmetricCiphertext OCTET STRING,
      macTag OCTET STRING
}
```

C.6 Syntax for Key Derivation Functions

This section provides ASN.1 syntax that may be used to encode input to the key derivation functions specified in Section 3.6. Note that the use of ASN.1 syntax for this purpose is optional — however the use of ASN.1 syntax in this scenario can help to ensure that the encoding of information fields is unambiguous.

The input to the key derivation function includes an octet string *SharedInfo*. *SharedInfo* may contain an encoding of `ASN1SharedInfo` as defined below.

```
ASN1SharedInfo ::= SEQUENCE {
      keyInfo AlgorithmIdentifier,
      entityUInfo [0] OCTET STRING OPTIONAL,
      entityVInfo [1] OCTET STRING OPTIONAL,
      suppPubInfo [2] OCTET STRING OPTIONAL,
      suppPrivInfo [3] OCTET STRING OPTIONAL
}
```

The components of type `ASN1SharedInfo` have the following meanings:

- `keyInfo` specifies the symmetric algorithm for which the derived key is to be used.
- `entityUInfo`, if present, specifies additional information about the scheme's initiator such as the entity's X.501 distinguished name, the entity's public key, etc.

- `entityVInfo`, if present, specifies additional information about the scheme's responder such as the entity's X.501 distinguished name, the entity's public key, etc.
- `suppPubInfo`, if present, specifies additional public information known to both entities involved in the operation of the scheme.
- `suppPrivInfo`, if present, specifies additional private information known to both entities involved in the operation of the scheme.

An example of the use of `ASN1SharedInfo` can be found in [3278].

C.7 Protocol Data Unit Syntax

The highest level types in an ASN.1 module, that is, those types not used in other types, are each known as a *Protocol Data Unit* (PDU), because, presumably these are the types that to be communicated in the protocol for which the ASN.1 module is to be applied. Lower level types, presumably, will not be protocol messages, but rather, only communicated as parts of the higher level types.

If the ASN.1 module is actually to be used in such a single protocol, then it makes sense to ensure that each PDU has a different tag. This makes BER decoding easier, for example. One way to achieve this is to define a single type, often including PDU in its name, of a `CHOICE` between the various next-to-highest level types. Tagging of the CHOICE type ensures that PDU type has a distinct tag.

Although it is not really the case the ASN.1 definitions and module used in SEC 1 are intended for use a single protocol, for completeness, a PDU definition as described above is given below.

```
SEC1-PDU ::= CHOICE {
      privateKey [0] ECPrivateKey,
      spki [1] SubjectPublicKeyInfo,
      ecdsa [2] ECDSA-Signature,
      ecies [3] ECIES-Ciphertext-Value,
      sharedinfo [4] ASN1SharedInfo,
      ...
}
```

It is emphasized that the expected use of the ASN.1 in this standard would be to import the ASN.1 definitions into another ASN.1 module, or to insert values of these types where consistent with ASN.1 types of other ASN.1 modules, rather than to use this higher level `CHOICE` type of PDU.

C.8 ASN.1 Module

The following comprises the ASN.1 module for all the items specified in this standard, including those that may have been defined in other modules.

```
SEC1-v1-9 {
      iso(1) identified-organization(3) certicom(132) module(1) ver(2)
}
DEFINITIONS EXPLICIT TAGS ::= BEGIN
      --
      -- EXPORTS ALL;
      --
FieldID { FIELD-ID:IOSet } ::= SEQUENCE { -- Finite field
      fieldType FIELD-ID.&id({IOSet}),
      parameters FIELD-ID.&Type({IOSet}{@fieldType})
}
FIELD-ID ::= TYPE-IDENTIFIER
FieldTypes FIELD-ID ::= {
      { Prime-p IDENTIFIED BY prime-field } |
      { Characteristic-two IDENTIFIED BY characteristic-two-field }
}
prime-field OBJECT IDENTIFIER ::= { id-fieldType 1 }
Prime-p ::= INTEGER -- Field of size p.
id-fieldType OBJECT IDENTIFIER ::= { ansi-X9-62 fieldType(1)}
ansi-X9-62 OBJECT IDENTIFIER ::= {
      iso(1) member-body(2) us(840) 10045
}
characteristic-two-field OBJECT IDENTIFIER ::= { id-fieldType 2 }
Characteristic-two ::= SEQUENCE {
      m INTEGER, -- Field size 2^m
      basis CHARACTERISTIC-TWO.&id({BasisTypes}),
      parameters CHARACTERISTIC-TWO.&Type({BasisTypes}{@basis})
}
CHARACTERISTIC-TWO ::= TYPE-IDENTIFIER
BasisTypes CHARACTERISTIC-TWO ::= {
      { NULL IDENTIFIED BY gnBasis } |
      { Trinomial IDENTIFIED BY tpBasis } |
      { Pentanomial IDENTIFIED BY ppBasis },
      ...
}
gnBasis OBJECT IDENTIFIER ::= { id-characteristic-two-basis 1 }
tpBasis OBJECT IDENTIFIER ::= { id-characteristic-two-basis 2 }
ppBasis OBJECT IDENTIFIER ::= { id-characteristic-two-basis 3 }
id-characteristic-two-basis OBJECT IDENTIFIER ::= {
      characteristic-two-field basisType(3)
}
Trinomial ::= INTEGER
Pentanomial ::= SEQUENCE {
      k1 INTEGER, -- k1 > 0
      k2 INTEGER, -- k2 > k1
```

```
      k3 INTEGER -- k3 > k2
}
FieldElement ::= OCTET STRING
ECDomainParameters{ECDOMAIN:IOSet} ::= CHOICE {
      specified SpecifiedECDomain,
      named ECDOMAIN.&id({IOSet}),
      implicitCA NULL
}
SpecifiedECDomain ::= SEQUENCE {
      version SpecifiedECDomainVersion(ecdpVer1 | ecdpVer2 | ecdpVer3, ...),
      fieldID FieldID {{FieldTypes}},
      curve Curve,
      base ECPoint,
      order INTEGER,
      cofactor INTEGER OPTIONAL,
      hash HashAlgorithm OPTIONAL,
      ...
}
SpecifiedECDomainVersion ::= INTEGER {
      ecdpVer1(1),
      ecdpVer2(2),
      ecdpVer3(3)
}
Curve ::= SEQUENCE {
      a FieldElement,
      b FieldElement,
      seed BIT STRING OPTIONAL
      -- Shall be present if used in SpecifiedECDomain
      -- with version equal to ecdpVer2 or ecdpVer3
}
ECPoint ::= OCTET STRING
ECDOMAIN ::= CLASS {
      &id OBJECT IDENTIFIER UNIQUE
}
WITH SYNTAX { ID &id }
SECGCurveNames ECDOMAIN::= {
      ...  -- named curves
}
HashAlgorithm ::= AlgorithmIdentifier {{ HashFunctions }}
HashFunctions ALGORITHM ::= {
       {OID sha-1 PARMS NULL } |
       {OID id-sha224 PARMS NULL } |
       {OID id-sha256 PARMS NULL } |
       {OID id-sha384 PARMS NULL } |
       {OID id-sha512 PARMS NULL } ,
```

```
      ...  -- Additional hash functions may be added in the future }
sha-1 OBJECT IDENTIFIER ::= {iso(1) identified-organization(3)
      oiw(14) secsig(3) algorithm(2) 26}
id-sha OBJECT IDENTIFIER ::= { joint-iso-itu-t(2) country(16) us(840)
      organization(1) gov(101) csor(3) nistalgorithm(4) hashalgs(2) }
id-sha224 OBJECT IDENTIFIER ::= { id-sha 4 }
id-sha256 OBJECT IDENTIFIER ::= { id-sha 1 }
id-sha384 OBJECT IDENTIFIER ::= { id-sha 2 }
id-sha512 OBJECT IDENTIFIER ::= { id-sha 3 }
SubjectPublicKeyInfo ::= SEQUENCE {
      algorithm AlgorithmIdentifier {{ECPKAlgorithms}} (WITH COMPONENTS
            {algorithm, parameters}) ,
      subjectPublicKey BIT STRING
}
AlgorithmIdentifier{ ALGORITHM:IOSet } ::= SEQUENCE {
      algorithm ALGORITHM.&id({IOSet}),
      parameters ALGORITHM.&Type({IOSet}{@algorithm}) OPTIONAL
}
ALGORITHM ::= CLASS {
      &id OBJECT IDENTIFIER UNIQUE,
      &Type OPTIONAL
}
      WITH SYNTAX { OID &id [PARMS &Type] }
ECPKAlgorithms ALGORITHM ::= {
      ecPublicKeyType  |
      ecPublicKeyTypeRestricted |
      ecPublicKeyTypeSupplemented  |
      {OID ecdh PARMS ECDomainParameters {{SECGCurveNames}}} |
      {OID ecmqv PARMS ECDomainParameters {{SECGCurveNames}}},
      ...
}
ecPublicKeyType ALGORITHM ::= {
      OID id-ecPublicKey PARMS ECDomainParameters {{SECGCurveNames}}
}
id-ecPublicKey OBJECT IDENTIFIER ::= { id-publicKeyType 1 }
id-publicKeyType OBJECT IDENTIFIER ::= { ansi-X9-62 keyType(2) }
ecPublicKeyTypeRestricted ALGORITHM ::= {
      OID id-ecPublicKeyTypeRestricted PARMS ECPKRestrictions
}
id-ecPublicKeyTypeRestricted OBJECT IDENTIFIER ::= {
      id-publicKeyType restricted(2) }
ECPKRestrictions ::= SEQUENCE {
      ecDomain ECDomainParameters {{ SECGCurveNames }},
      eccAlgorithms ECCAlgorithms
}
```

```
ECCAlgorithms ::= SEQUENCE OF ECCAlgorithm
ECCAlgorithm ::= AlgorithmIdentifier {{ECCAlgorithmSet}}
ecPublicKeyTypeSupplemented ALGORITHM ::= {
      OID id-ecPublicKeyTypeSupplemented PARMS ECPKSupplements
}
secg-scheme OBJECT IDENTIFIER ::= { iso(1)
      identified-organization(3) certicom(132) schemes(1) }
id-ecPublicKeyTypeSupplemented OBJECT IDENTIFIER ::= {
      secg-scheme supplementalPoints(0) }
ECPKSupplements ::= SEQUENCE {
      ecDomain ECDomainParameters {{ SECGCurveNames }},
      eccAlgorithms ECCAlgorithms,
      eccSupplements ECCSupplements }
ECCSupplements ::= CHOICE {
      namedMultiples [0] NamedMultiples,
      specifiedMultiples [1] SpecifiedMultiples
}
NamedMultiples ::= SEQUENCE {
      multiples OBJECT IDENTIFIER,
      points SEQUENCE OF ECPoint }
SpecifiedMultiples ::= SEQUENCE OF SEQUENCE {
      multiple INTEGER,
      point ECPoint }
ECPrivateKey ::= SEQUENCE {
      version INTEGER { ecPrivkeyVer1(1) } (ecPrivkeyVer1),
      privateKey OCTET STRING,
      parameters [0] ECDomainParameters {{ SECGCurveNames }} OPTIONAL,
      publicKey [1] BIT STRING OPTIONAL
}
ecdsa-with-SHA1 OBJECT IDENTIFIER ::= { id-ecSigType sha1(1)}
ecdsa-with-Recommended OBJECT IDENTIFIER ::= { id-ecSigType recommended(2) }
ecdsa-with-Specified OBJECT IDENTIFIER ::= { id-ecSigType specified(3)}
ecdsa-with-Sha224 OBJECT IDENTIFIER ::= { id-ecSigType specified(3) 1 }
ecdsa-with-Sha256 OBJECT IDENTIFIER ::= { id-ecSigType specified(3) 2 }
ecdsa-with-Sha384 OBJECT IDENTIFIER ::= { id-ecSigType specified(3) 3 }
ecdsa-with-Sha512 OBJECT IDENTIFIER ::= { id-ecSigType specified(3) 4 }
id-ecSigType OBJECT IDENTIFIER ::= { ansi-X9-62 signatures(4) }
ECDSAAlgorithmSet ALGORITHM ::= {
      {OID ecdsa-with-SHA1 PARMS NULL} |
      {OID ecdsa-with-Recommended PARMS NULL} |
      {OID ecdsa-with-Specified PARMS HashAlgorithm } |
      {OID ecdsa-with-Sha224 PARMS NULL} |
      {OID ecdsa-with-Sha256 PARMS NULL} |
      {OID ecdsa-with-Sha384 PARMS NULL} |
      {OID ecdsa-with-Sha512 PARMS NULL} ,
```

```
      ...  -- More algorithms need to be added
}
ECCAlgorithmSet ALGORITHM ::= {
      ECDSAAlgorithmSet |
      ECDHAlgorithmSet |
      ECMQVAlgorithmSet |
      ECIESAlgorithmSet |
      ECWKTAlgorithmSet ,
      ...
}
ECDHAlgorithmSet ALGORITHM ::= {
      {OID dhSinglePass-stdDH-sha1kdf PARMS NULL} |
      {OID dhSinglePass-cofactorDH-sha1kdf PARMS NULL} |
      {OID dhSinglePass-cofactorDH-recommendedKDF} |
      {OID dhSinglePass-cofactorDH-specifiedKDF PARMS KeyDerivationFunction} |
      {OID ecdh} |
      {OID dhSinglePass-stdDH-sha256kdf-scheme} |
      {OID dhSinglePass-stdDH-sha384kdf-scheme} |
      {OID dhSinglePass-stdDH-sha224kdf-scheme} |
      {OID dhSinglePass-stdDH-sha512kdf-scheme} |
      {OID dhSinglePass-cofactorDH-sha256kdf-scheme} |
      {OID dhSinglePass-cofactorDH-sha384kdf-scheme} |
      {OID dhSinglePass-cofactorDH-sha224kdf-scheme} |
      {OID dhSinglePass-cofactorDH-sha512kdf-scheme} ,
      ...  -- Future combinations may be added
}
ECMQVAlgorithmSet ALGORITHM ::= {
      {OID mqvSinglePass-sha1kdf} |
      {OID mqvSinglePass-recommendedKDF} |
      {OID mqvSinglePass-specifiedKDF PARMS KeyDerivationFunction} |
      {OID mqvFull-sha1kdf} |
      {OID mqvFull-recommendedKDF} |
      {OID mqvFull-specifiedKDF PARMS KeyDerivationFunction} |
      {OID ecmqv} |
      {OID mqvSinglePass-sha256kdf-scheme } |
      {OID mqvSinglePass-sha384kdf-scheme } |
      {OID mqvSinglePass-sha224kdf-scheme } |
      {OID mqvSinglePass-sha512kdf-scheme } |
      {OID mqvFull-sha256kdf-scheme } |
      {OID mqvFull-sha384kdf-scheme } |
      {OID mqvFull-sha224kdf-scheme } |
      {OID mqvFull-sha512kdf-scheme } ,
      ...  -- Future combinations may be added
}
x9-63-scheme OBJECT IDENTIFIER ::= { iso(1) member-body(2)
```

```
      us(840) ansi-x9-63(63) schemes(0) }
dhSinglePass-stdDH-sha1kdf OBJECT IDENTIFIER ::= {x9-63-scheme 2}
dhSinglePass-cofactorDH-sha1kdf OBJECT IDENTIFIER ::= {x9-63-scheme 3}
mqvSinglePass-sha1kdf OBJECT IDENTIFIER ::= {x9-63-scheme 16}
mqvFull-sha1kdf OBJECT IDENTIFIER ::= {x9-63-scheme 17}
dhSinglePass-cofactorDH-recommendedKDF OBJECT IDENTIFIER ::= {secg-scheme 1}
dhSinglePass-cofactorDH-specifiedKDF OBJECT IDENTIFIER ::= {secg-scheme 2}
ecdh OBJECT IDENTIFIER ::= {secg-scheme 12}
dhSinglePass-stdDH-sha256kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 11 1}
dhSinglePass-stdDH-sha384kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 11 2}
dhSinglePass-stdDH-sha224kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 11 0}
dhSinglePass-stdDH-sha512kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 11 3}
dhSinglePass-cofactorDH-sha256kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 14 1}
dhSinglePass-cofactorDH-sha384kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 14 2}
dhSinglePass-cofactorDH-sha224kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 14 0}
dhSinglePass-cofactorDH-sha512kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 14 3}
mqvSinglePass-recommendedKDF OBJECT IDENTIFIER ::= {secg-scheme 3}
mqvSinglePass-specifiedKDF OBJECT IDENTIFIER ::= {secg-scheme 4}
mqvFull-recommendedKDF OBJECT IDENTIFIER ::= {secg-scheme 5}
mqvFull-specifiedKDF OBJECT IDENTIFIER ::= {secg-scheme 6}
ecmqv OBJECT IDENTIFIER ::= {secg-scheme 13}
mqvSinglePass-sha256kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 15 1}
mqvSinglePass-sha384kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 15 2}
mqvSinglePass-sha224kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 15 0}
mqvSinglePass-sha512kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 15 3}
mqvFull-sha256kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 16 1}
mqvFull-sha384kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 16 2}
mqvFull-sha224kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 16 0}
mqvFull-sha512kdf-scheme OBJECT IDENTIFIER ::= {secg-scheme 16 3}
KeyDerivationFunction ::= AlgorithmIdentifier {{KDFSet}}
KDFSet ALGORITHM ::= {
      {OID x9-63-kdf PARMS HashAlgorithm } |
      {OID nist-concatenation-kdf PARMS HashAlgorithm } |
      {OID tls-kdf PARMS HashAlgorithm } |
      {OID ikev2-kdf PARMS HashAlgorithm } ,
      ...  -- Future combinations may be added
}
x9-63-kdf OBJECT IDENTIFIER ::= {secg-scheme 17 0}
nist-concatenation-kdf OBJECT IDENTIFIER ::= {secg-scheme 17 1}
tls-kdf OBJECT IDENTIFIER ::= {secg-scheme 17 2}
ikev2-kdf OBJECT IDENTIFIER ::= {secg-scheme 17 3}
ECIESAlgorithmSet ALGORITHM ::= {
      {OID ecies-recommendedParameters} |
      {OID ecies-specifiedParameters PARMS ECIESParameters} ,
      ...  -- Future combinations may be added
```

```
}
ecies-recommendedParameters OBJECT IDENTIFIER ::= {secg-scheme 7}
ecies-specifiedParameters OBJECT IDENTIFIER ::= {secg-scheme 8}
ECIESParameters ::= SEQUENCE {
      kdf [0] KeyDerivationFunction OPTIONAL,
      sym [1] SymmetricEncryption OPTIONAL,
      mac [2] MessageAuthenticationCode OPTIONAL
}
SymmetricEncryption ::= AlgorithmIdentifier {{SYMENCSet}}
MessageAuthenticationCode ::= AlgorithmIdentifier {{MACSet}}
SYMENCSet ALGORITHM ::= {
      { OID xor-in-ecies } |
      { OID tdes-cbc-in-ecies } |
      { OID aes128-cbc-in-ecies } |
      { OID aes192-cbc-in-ecies } |
      { OID aes256-cbc-in-ecies } |
      { OID aes128-ctr-in-ecies } |
      { OID aes192-ctr-in-ecies } |
      { OID aes256-ctr-in-ecies } ,
      ...  -- Future combinations may be added
}
MACSet ALGORITHM ::= {
      { OID hmac-full-ecies PARMS HashAlgorithm} |
      { OID hmac-half-ecies PARMS HashAlgorithm} |
      { OID cmac-aes128-ecies } |
      { OID cmac-aes192-ecies } |
      { OID cmac-aes256-ecies } ,
      ...  -- Future combinations may be added
}
xor-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 18 }
tdes-cbc-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 19 }
aes128-cbc-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 20 0 }
aes192-cbc-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 20 1 }
aes256-cbc-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 20 2 }
aes128-ctr-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 21 0 }
aes192-ctr-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 21 1 }
aes256-ctr-in-ecies OBJECT IDENTIFIER ::= {secg-scheme 21 2 }
hmac-full-ecies OBJECT IDENTIFIER ::= {secg-scheme 22 }
hmac-half-ecies OBJECT IDENTIFIER ::= {secg-scheme 23 }
cmac-aes128-ecies OBJECT IDENTIFIER ::= {secg-scheme 24 0 }
cmac-aes192-ecies OBJECT IDENTIFIER ::= {secg-scheme 24 1 }
cmac-aes256-ecies OBJECT IDENTIFIER ::= {secg-scheme 24 2 }
ECWKTAlgorithmSet ALGORITHM ::= {
      {OID ecwkt-recommendedParameters} |
      {OID ecwkt-specifiedParameters PARMS ECWKTParameters} ,
```

```
      ...  -- Future combinations may be added
}
ecwkt-recommendedParameters OBJECT IDENTIFIER ::= {secg-scheme 9}
ecwkt-specifiedParameters OBJECT IDENTIFIER ::= {secg-scheme 10}
ECWKTParameters ::= SEQUENCE {
      kdf [0] KeyDerivationFunction OPTIONAL,
      wrap [1] KeyWrapFunction OPTIONAL
}
KeyWrapFunction ::= AlgorithmIdentifier {{KeyWrapSet}}
KeyWrapSet ALGORITHM ::= {
      { OID aes128-key-wrap } |
      { OID aes192-key-wrap } |
      { OID aes256-key-wrap } ,
      ...  -- Future combinations may be added
}
aes128-key-wrap OBJECT IDENTIFIER ::= {secg-scheme 25 0 }
aes192-key-wrap OBJECT IDENTIFIER ::= {secg-scheme 25 1 }
aes256-key-wrap OBJECT IDENTIFIER ::= {secg-scheme 25 2 }
ECDSA-Signature ::= CHOICE {
      two-ints-plus ECDSA-Sig-Value,
      point-int [0] ECDSA-Full-R,
      ...  -- Future representations may be added
}
ECDSA-Sig-Value ::= SEQUENCE {
      r INTEGER,
      s INTEGER,
      a INTEGER OPTIONAL,
      y CHOICE { b BOOLEAN, f FieldElement } OPTIONAL
}
ECDSA-Full-R ::= SEQUENCE {
      r ECPoint,
      s INTEGER
}
ECIES-Ciphertext-Value ::= SEQUENCE {
      ephemeralPublicKey ECPoint,
      symmetricCiphertext OCTET STRING,
      macTag OCTET STRING
}
ASN1SharedInfo ::= SEQUENCE {
      keyInfo AlgorithmIdentifier,
      entityUInfo [0] OCTET STRING OPTIONAL,
      entityVInfo [1] OCTET STRING OPTIONAL,
      suppPubInfo [2] OCTET STRING OPTIONAL,
      suppPrivInfo [3] OCTET STRING OPTIONAL
}
```

```
SEC1-PDU ::= CHOICE {
      privateKey [0] ECPrivateKey,
      spki [1] SubjectPublicKeyInfo,
      ecdsa [2] ECDSA-Signature,
      ecies [3] ECIES-Ciphertext-Value,
      sharedinfo [4] ASN1SharedInfo,
      ...
}
END
```

D References

[46-2] National Institute of Standards and Technology. *Data Encryption Standard*, Federal Information Processing Standard 46-2, 1993. Withdrawn. csrc.nist.gov/groups/ST/toolkit/block_ciphers.html.

[180-1] ———. *Secure Hash Standard*, Federal Information Processing Standard 180-1, 1995.

[180-2] ———. *Secure Hash Standard (Change Notice)*, Federal Information Processing Standard 180-2, Feb. 2004. csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

[186] ———. *Digital Signature Standard*, Federal Information Processing Standard 186, 1993.

[186-2] ———. *Digital Signature Standard (Change Notice)*, Federal Information Processing Standard 186-2, Oct. 2001. csrc.nist.gov/groups/ST/toolkit/digital_signatures.html.

[186-3] ———. *Digital Signature Standard (Change Notice)*, Federal Information Processing Standard 186-3, 2005. Draft csrc.nist.gov/groups/ST/toolkit/digital_signatures.html.

[197] ———. *Advanced Encryption Standard (Change Notice)*, Federal Information Processing Standard 197, Oct. 2001. csrc.nist.gov/groups/ST/toolkit/block_ciphers.html.

[198] ———. *The Keyed-Hash Message Authentication Code (HMAC)*, Federal Information Processing Standard 198, Mar. 2002. http://csrc.nist.gov/groups/ST/toolkit/message_auth.html.

[800-38A] M. Dworkin. *Recommendation for Block Cipher Modes of Operation*, Special Publication 800-38A. National Institute of Standards and Technology, Dec. 2001. csrc.nist.gov/groups/ST/toolkit/BCM/index.html.

[800-38B] ———. *Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication*, Special Publication 800-38B. National Institute of Standards and Technology, May 2005. csrc.nist.gov/groups/ST/toolkit/BCM/index.html.

[800-56A] E. Barker, D. Johnson and M. Smid. *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*, Special Publication 800-56A. National Institute of Standards and Technology, Mar. 2007. csrc.nist.gov/groups/ST/toolkit/key_management.html.

[800-90] E. Barker and J. Kelsey. *Recommendation for Random Number Generation Using Deterministic Bit Generators*, Special Publication 800-90. National Institute of Standards and Technology, Mar. 2007. csrc.nist.gov/groups/ST/toolkit/random_number.html.

[800-106] Q. Dang. *Randomized Hashing Digital Signatures*, Special Publication 800-106. National Institute of Standards and Technology, Jul. 2007. csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

[1363] Institute of Electrical and Electronics Engineers. *Specifications for Public-Key Cryptography*, IEEE Standard 1363-2000, Aug. 2000. http://standards.ieee.org/catalog/olis/busarch.html.

[1363A] ———. *Specifications for Public-Key Cryptography — Amendment 1: Additional Techniques*, IEEE Standard 1363A-2004, Oct. 2004. http://standards.ieee.org/catalog/olis/busarch.html.

[2104] H. Krawczyk, M. Bellare and R. Canetti. *HMAC: Keyed Hashing for Message Authentication*, Request For Comments 2104. Internet Engineering Task Force, 1997. tools.ietf.org/html/rfc2104.

[2246] T. Dierks and C. Allen (eds.). *The TLS Protocol, Version 1.0*, Request For Comments 2246, Jan. 1999. tools.ietf.org/html/rfc2246.

[2409] D. Harkins and D. Carrel. *The Internet Key Exchange*, Request For Comments 2409. Internet Engineering Task Force, 1998. tools.ietf.org/html/rfc2409.

[2630] R. Housley. *Cryptographic Message Syntax*, Request For Comments 2630. Internet Engineering Task Force, Jun. 1999. tools.ietf.org/html/rfc2630.

[3278] S. Blake-Wilson, D. R. L. Brown and P. Lambert. *Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)*, Request For Comments 3278. Internet Engineering Task Force, Apr. 2002. tools.ietf.org/html/rfc3278.

[3279] L. Bassham, R. Housley and W. Polk. *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, Request For Comments 3279. Internet Engineering Task Force, Apr. 2002. tools.ietf.org/html/rfc3279.

[3394] J. Schaad and R. Housley. *Advanced Encryption Standard (AES) Key Wrap Algorithm*, Request For Comments 3394. Internet Engineering Task Force, Sep. 2002. tools.ietf.org/html/rfc3394.

[4306] C. Kaufman (ed.). *Internet Key Exchange (IKEv2) Protocol*, Request For Comments 4306, Dec. 2005. tools.ietf.org/html/rfc4306.

[4492] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk and B. Möller. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*, Request For Comments 4492. Internet Engineering Task Force, May 2006. tools.ietf.org/html/rfc4492.

[4753] D. E. Fu and J. A. Solinas. *ECP Groups For IKE and IKEv2*, Request For Comments 4753. Internet Engineering Task Force, Jan. 2007. tools.ietf.org/html/rfc4753.

[5480] S. Turner, D. R. L. Brown, K. Yiu, R. Housley and T. Polk. *Elliptic Curve Cryptography Subject Public Key Information*, Request For Comments 5480. Internet Engineering Task Force, Mar. 2009. `tools.ietf.org/html/rfc5480`.

[14888-3] International Standards Organization. *Information Technology — Security Techniques — Digital Signatures with Appendix: — Part 3: Certificate-Based Mechanisms*, International Standard 14888-3, Dec. 1998.

[15946-1] ———. *Information Technology — Security Techniques — Cryptographic Techniques Based on Elliptic Curves: — Part 1: General*, International Standard 15946-1, Dec. 2002.

[15946-2] ———. *Information Technology — Security Techniques — Cryptographic Techniques Based on Elliptic Curves — Part 2: Digital Signatures*, International Standard 15946-2, Dec. 2002.

[15946-3] ———. *Information Technology — Security Techniques — Cryptographic Techniques Based on Elliptic Curves — Part 3: Key Establishment*, International Standard 15946-3, Dec. 2002.

[18033-2] V. Shoup (ed.). *Encryption Algorithms — Part 2: Asymmetric Ciphers*, International Standard 18033-2, May 2006. `http://shoup.net/iso`.

[ASC04] M. Dworkin. *Request for Review of Key Wrap Algorithms.* ASC X9, Nov. 2004. `http://eprint.iacr.org/2004/340`.

[GEC 2] Standards for Efficient Cryptography Group. *GEC 2: Test Vectors for SEC 1*, Sep. 1999. Working Draft. `http://www.secg.org`.

[Int06a] D. R. L. Brown. *Additional ECC Groups For IKE and IKEv2.* Internet Engineering Task Force, Oct. 2006. Expired. `http://tools.ietf.org/html/draft-ietf-ipsec-ike-ecc-groups-10`.

[Int06b] ———. *Internet-Draft: Additional Algorithms and Identifiers for Use of ECC with PKIX.* Internet Engineering Task Force, Oct. 2006. Expired `http://tools.ietf.org/html/draft-ietf-pkix-ecc-pkalgs-03`.

[NESSIE] B. Preneel, A. Biryukov, C. D. Cannière, S. B. Örs, E. Oswald, B. V. Rompay, L. Granboulan, E. Dottax, G. Martinet, S. Murphy, A. Dent, R. Shipsey, C. Swart, J. White, M. Dichtl, S. Pyka, M. Schafheutle, P. Serf, E. Biham, E. Barkan, Y. Braziler, O. Dunkelman, V. Furman, D. Kenigsberg, J. Stolin, J.-J. Quisquater, M. Ciet, F. Sica, H. Raddum, L. Knudsen and M. Parker. *New European Schemes for Signatures, Integrity and Encryption*, IST-1999-12324. Information Society Technologies (IST) Programme, Apr. 2004. Draft Version 0.15 (beta), `http://www.cryptonessie.org`.

[Nat99] National Institute of Standards and Technology. *Recommended Elliptic Curves for Federal Government Use*, Jul. 1999. `http://csrc.nist.gov/encryption`.

[Nat01] M. Dworkin. *AES Key Wrap Specification*. National Institute of Standards and Technology, Nov. 2001. `http://csrc.nist.gov/groups/ST/toolkit/key_management.html`.

[PKCS8] RSA Laboratories. *PKCS #8: Private-Key Information Syntax Standard*, Nov. 1993. `www.rsa.com/rsalabs`.

[SEC 1] Standards for Efficient Cryptography Group. *SEC 1: Elliptic Curve Cryptography*, Sep. 2000. Version 1.0. `http://www.secg.org/download/aid-385/sec1_final.pdf`.

[SEC 2] ———. *SEC 2: Recommended Elliptic Curve Domain Parameters*, Sep. 2000. Version 1.0. `http://www.secg.org/download/aid-386/sec2_final.pdf`.

[WTLS] Wireless Application Forum. *WAP WTLS: Wireless Application Protocol Wireless Transport Layer Security Specification*, Feb. 1999.

[X9.52] American National Standards Institute. *Triple Data Encryption: Modes of Operation*, American National Standard X9.52-1998, 1998. `http://webstore.ansi.org/ansidocstore`.

[X9.62a] ———. *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, American National Standard X9.62-1998, 1998. Obsoleted by [X9.62b].

[X9.62b] ———. *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, American National Standard X9.62-2005, 2005. `http://webstore.ansi.org/ansidocstore`.

[X9.63] ———. *Public-Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*, American National Standard X9.63-2001, 2001. `http://webstore.ansi.org/ansidocstore`.

[X9.71] ———. *Keyed Hash Message Authentication Code*, American National Standard X9.71-2001, 2001. Not currently available.

[X9.82] ———. *Random Number Generation*, Draft American National Standard X9.82, 2005. Tentative organization: Part 1: Overview; Part 2: Entropy Sources; Part 3: Deterministic Algorithms; Part 4: Complete Systems.

[X9.92] ———. *Public-Key Cryptography for the Financial Services Industry: Digital Signature Algorithms Providing Partial Message Recovery: Part 1: Elliptic Curve Pintsov-Vanstone Signatures (ECPVS)*, Draft American National Standard X9.92-2002, 2002.

[X9.102] ———. *Symmetric Key Cryptography for the Financial Services Industry: Part 1: Wrapping of Keys and Associated Data*, Draft American National Standard X9.102-2003, 2003. Draft.

[X.681] International Telecommunications Union. *ITU-T Recommendation X.681: Information Technology — Abstract Syntax Notation One (ASN.1): Information Object Specification*, Jul. 1994. Equivalent to ISO/IEC 8824-2.

[X.690] ———. *ITU-T Recommendation X.690: Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER)*, Dec. 1997. Equivalent to ISO/IEC 8825-1.

[Atk92] A. O. L. Atkin. *The number of points on an elliptic curve modulo a prime.* Series of emails to the NMBRTHRY mailing list, 1992.

[ABG+05] A. Antipa, D. R. L. Brown, R. P. Gallant, R. Lambert, R. Struik and S. A. Vanstone. *Accelerated verification of ECDSA signatures.* In B. Preneel and S. Tavares (eds.), *Selected Areas in Cryptography: SAC 2005*, Lecture Notes in Computer Science 3897, pp. 307–318. Springer, Aug. 2005.

[ABM+03] A. Antipa, D. R. L. Brown, A. J. Menezes, R. Struik and S. A. Vanstone. *Validation of elliptic curve public keys.* In Y. G. Desmedt (ed.), *Public Key Cryptography — PKC 2003*, Lecture Notes in Computer Science 2567, pp. 211–223. International Association for Cryptologic Research, Springer, Jan. 2003.

[ABMV93] G. Agnew, T. Beth, R. Mullin and S. A. Vanstone. *Arithmetic operations in $GF(2^m)$.* Journal of Cryptology, **6**:3–13, 1993.

[ABR01a] M. Abdalla, M. Bellare and P. Rogaway. *DHIES: An encryption scheme based on the Diffie-Hellman problem.* `www-cse.ucsd.edu/users/mihir`, Sep. 2001. Full version of [ABR01b].

[ABR01b] ———. *The oracle Diffie-Hellman assumptions and an analysis of DHIES.* In Naccache [Nac01], pp. 143–158.

[AMOV91] G. B. Agnew, R. C. Mullin, I. M. Onyszchuk and S. A. Vanstone. *An implementation for a fast public-key cryptosystem.* Journal of Cryptology, **3**:63–79, 1991.

[AMV93] G. Agnew, R. Mullin and S. A. Vanstone. *An implementation of elliptic curve cryptosystems over $\mathbb{F}_{2^{155}}$.* IEEE Journal on Selected Areas in Communications, **11**:804–813, 1993.

[Ble01] D. Bleichenbacher. *On the generation of DSS one-time keys.* preprint, 2001.

[Bon98] D. Boneh. *The decision Diffie-Hellman problem.* In J. P. Buhler (ed.), *Algorithmic Number Theory III*, Lecture Notes in Computer Science 1423, pp. 48–63. Springer, Jun. 1998.

[Bro01] D. R. L. Brown. *A conditional security analysis of the elliptic curve digital signature algorithm.* In *The 5th Workshop on Elliptic Curve Cryptography (ECC 2001)*. Oct. 2001. `http://www.cacr.math.uwaterloo.ca/conferences/2001/ecc/brown.ps`.

[Bro05a] ———. *Generic groups, collision resistance, and ECDSA.* Designs, Codes and Cryptography, **35**:119–152, 2005. `http://eprint.iacr.org/2002/026`.

[Bro05b] ———. *On the provable security of ECDSA*. In Blake et al. [BSS05], pp. 21–40. Chapter II.

[BCK98] M. Bellare, R. Canetti and H. Krawczyk. *A modular approach to the design and analysis of authentication and key exchange protocols*. In *30th Annual Symposium on the Theory of Computing*. 1998.

[BDL97] D. Boneh, R. A. DeMillo and R. J. Lipton. *On the importance of checking cryptographic protocols for faults*. In W. Fumy (ed.), *Advances in Cryptology – EUROCRYPT '97*, Lecture Notes in Computer Science 1233, pp. 37–51. International Association for Cryptologic Research, Springer, May 1997.

[BG04a] D. R. L. Brown and R. P. Gallant. *The static Diffie-Hellman problem*. ePrint 2004/306, International Association for Cryptologic Research, 2004. `http://eprint.iacr.org/2004/306`.

[BG04b] ———. *The static Diffie-Hellman problem*. CACR 2004/04, Centre for Applied Cryptologic Research, 2004. Alternate version of [BG04a] at `http://www.cacr.math.uwaterloo.ca/techreports/2004/cacr2004-10.ps`.

[BG07] D. R. L. Brown and K. Gjøsteen. *A security analysis of the NIST SP 800-90 elliptic curve random number generator*. In A. J. Menezes (ed.), *Advances in Cryptology — CRYPTO 2007*, Lecture Notes in Computer Science 4622, pp. 466–481. International Association for Cryptologic Research, Springer, Aug. 2007. `http://eprint.iacr.org/2007/048`.

[BGM97] M. Bellare, S. Goldwasser and D. Micciancio. *"Pseudo-random" number generation within cryptographic algorithms: The DSS case*. In Kaliski [Kal97], pp. 277–291.

[BJ01] D. R. L. Brown and D. B. Johnson. *Formal security proofs for a signature scheme with partial message recovery*. In Naccache [Nac01], pp. 126–142. `http://www.cacr.math.uwaterloo.ca/techreports/2000/corr2000-39.pdf`.

[BL96] D. Boneh and R. J. Lipton. *Algorithms for black-box fields and their application to cryptography*. In Koblitz [Kob96], pp. 283–297.

[BR97] M. Bellare and P. Rogaway. *Minimizing the use of random oracles in authenticated encryption schemes*. In Y. Han, T. Okamoto and S. Qing (eds.), *Information and Communications Security*, Lecture Notes in Computer Science 1334, pp. 1–16. Springer, Nov. 1997.

[BSS99] I. F. Blake, G. Seroussi and N. P. Smart. *Elliptic Curves in Cryptography*. London Mathematical Society Lecture Notes 265. Cambridge University Press, 1999.

[BSS05] I. F. Blake, G. Seroussi and N. P. Smart (eds.). *Advances in Elliptic Curve Cryptography*. London Mathematical Society Lecture Notes 317. Cambridge University Press, 2005.

[BV96] D. Boneh and R. Venkatesan. *Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes*. In Koblitz [Kob96], pp. 129–142.

[BWJM97] S. Blake-Wilson, D. B. Johnson and A. J. Menezes. *Key agreement protocols and their security analysis*. In M. Darnell (ed.), *Cryptography and Coding*, Lecture Notes in Computer Science 1355, pp. 30–45. Springer, Dec. 1997.

[BWM99] S. Blake-Wilson and A. J. Menezes. *Unknown key-share attacks on the station-to-station (STS) protocol*. In H. Imai and Y. Zheng (eds.), *Public Key Cryptography — PKC '99*, Lecture Notes in Computer Science 1560, pp. 154–170. International Association for Cryptologic Research, Springer, 1999.

[Che06] J. H. Cheon. *Security analysis of the strong Diffie-Hellman problem*. In S. Vaudenay (ed.), *Advances in Cryptology – EUROCRYPT 2006*, Lecture Notes in Computer Science 4004, pp. 1–11. International Association for Cryptologic Research, Springer, May 2006. `http://www.math.snu.ac.kr/~jhcheon/publications/2006/Eurocrypt_Cheon_LNCS.pdf`.

[CFA+06] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2006.

[CH98] M. Chen and E. Hughes. *Protocol failures related to order of encryption and signature: Computation of discrete logarithms in RSA groups*. In C. Boyd and E. Dawson (eds.), *Information Security and Privacy — ACISP '98*, Lecture Notes in Computer Science 1438. Jul. 1998.

[CS98] R. Cramer and V. Shoup. *A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack*. In H. Krawczyk (ed.), *Advances in Cryptology — CRYPTO '98*, Lecture Notes in Computer Science 1462, pp. 13–25. International Association for Cryptologic Research, Springer, Aug. 1998. `http://www.shoup.net/papers/cs.pdf`.

[CS01] ———. *Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack*. ePrint 2001/108, International Association for Cryptologic Research, Dec. 2001. `http://eprint.iacr.org/2004/306`.

[Den05] A. W. Dent. *Proofs of security for ECIES*. In Blake et al. [BSS05], pp. 41–66. Chapter III.

[DvOW92] W. Diffie, P. C. van Oorschot and M. J. Wiener. *Authentication and authenticated key exchanges*. Designs, Codes and Cryptography, **2**:107–125, 1992.

[DH76] W. Diffie and M. E. Hellman. *New directions in cryptography*. IEEE Transactions on Information Theory, **IT-22**(6):644–654, Nov. 1976.

[Elk98] N. D. Elkies. *Elliptic and modular curves over finite fields and related computational issues*. In *Computational Perspectives in Number Theory*, pp. 21–76. American Mathematical Society International Press, 1998.

[ElG85] T. ElGamal. *A public key cryptosystem and a signature scheme based on discrete logarithms.* IEEE Transactions on Information Theory, **IT-31**:469–472, 1985.

[ECC99] *ECC Challenge.* Details at `http://www.certicom.com`, 1999.

[FR94] G. Frey and H.-G. Rück. *A remark concerning m-divisibility and the discrete logarithm problem in the divisor class group of curves.* Mathematics of Computation, **62**:865–874, 1994.

[Gal05] S. D. Galbraith. *Pairings.* In Blake et al. [BSS05], pp. 183–213. Chapter IX.

[GHS02] P. Gaudry, F. Hess and N. P. Smart. *Constructive and destructive facets of Weil descent on elliptic curves.* J. of Cryptology, **15**:19–46, 2002. `http://www.loria.fr/~gaudry/publis/weildesc_vZ.ps.gz`.

[GLV00] R. P. Gallant, R. Lambert and S. A. Vanstone. *Improving the parallelized Pollard lambda search on binary anomalous curves.* Mathematics of Computation, **69**:1699–1705, 2000.

[GM05] S. D. Galbraith and A. J. Menezes. *Algebraic curves and cryptography.* Finite Fields and Their Applications, **11**(3):544–577, 2005.

[GS99] S. D. Galbraith and N. P. Smart. *A cryptographic application of the Weil descent.* In M. Walker (ed.), *Cryptography and Coding*, Lecture Notes in Computer Science 1746, pp. 191–200. Springer, Dec. 1999.

[Hes05] F. Hess. *Weil descent attacks.* In Blake et al. [BSS05], pp. 151–180. Chapter VIII.

[Hit07] L. Hitt. *On the minimal embedding field.* In T. Takagi, T. Okamoto, E. Okamoto and T. Okamoto (eds.), *Pairing 2007*, Lecture Notes in Computer Science 4575, pp. 294–301. Springer, Jul. 2007. `http://eprint.iacr.org/2006/415`.

[HGS01] N. Howgrave-Graham and N. P. Smart. *Lattice attacks on digital signature schemes.* Designs, Codes and Cryptography, **23**:283–290, 2001.

[HMV04] D. Hankerson, A. J. Menezes and S. A. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2004.

[Joh96] D. B. Johnson. *Diffie-Hellman key agreement small subgroup attack*, Jul. 16 1996. Contribution to X9F1 by Certicom.

[Jun93] D. Jungnickel. *Finite Fields: Structure and Arithmetics.* B. I. Wissenschaftsverlag, Mannheim, 1993.

[JMS01] M. Jacobson, A. J. Menezes and A. Stein. *Solving elliptic curve discrete logarithm problems using Weil descent.* J. of the Ramanujan Mathematical Society, **16**:231–260, 2001. `http://eprint.iacr.org/2001/041`.

[JMV01] D. B. Johnson, A. J. Menezes and S. A. Vanstone. *The elliptic curve digital signature algorithm (ECDSA).* International Journal on Information Security, **1**(1):36–63, Feb. 2001. `http://www.cacr.math.uwaterloo.ca/techreports/1999/corr99-34.pdf`.

[Kal97] B. S. Kaliski (ed.). *Advances in Cryptology — CRYPTO '97*, Lecture Notes in Computer Science 1294. International Association for Cryptologic Research, Springer, Aug. 1997.

[Kal98] B. Kaliski. *MQV vulnerability.* Posting to ANSI X9F1 and IEEE P1363 newsgroups, 1998.

[Knu81] D. Knuth. *The Art of Computer Programming — Seminumerical Algorithms*, vol. 2. Addison-Wesley, second edn., 1981.

[Kob87] N. Koblitz. *Elliptic curve cryptosystems.* Mathematics of Computation, **48**:203–209, 1987.

[Kob91] ———. *CM-curves with good cryptographic properties.* In J. Feigenbaum (ed.), *Advances in Cryptology — CRYPTO '91*, Lecture Notes in Computer Science 576, pp. 279–287. International Association for Cryptologic Research, Springer, Aug. 1991.

[Kob94] ———. *A Course in Number Theory and Cryptography.* Springer, second edn., 1994.

[Kob96] N. Koblitz (ed.). *Advances in Cryptology — CRYPTO '96*, Lecture Notes in Computer Science 1109. International Association for Cryptologic Research, Springer, Aug. 1996.

[Koc96] P. Kocher. *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems.* In Koblitz [Kob96], pp. 104–113.

[KJJ99] P. Kocher, J. Jaffe and B. Jun. *Differential power analysis.* In M. J. Wiener (ed.), *Advances in Cryptology — CRYPTO '99*, Lecture Notes in Computer Science 1666, pp. 388–397. International Association for Cryptologic Research, Springer, Aug. 1999.

[KM05] N. Koblitz and A. J. Menezes. *Pairing-based cryptography at high security levels.* In N. P. Smart (ed.), *Coding and Cryptography*, Lecture Notes in Computer Science 3796, pp. 13–36. Springer, Dec. 2005. `http://eprint.iacr.org/2005/076`.

[LL97] C. H. Lim and P. J. Lee. *A key recovery attack on discrete log-based schemes using a prime order subgroup.* In Kaliski [Kal97], pp. 249–263.

[LMQ+98] L. Law, A. J. Menezes, M. Qu, J. A. Solinas and S. A. Vanstone. *An efficient protocol for authenticated key agreement.* CORR 98-05, Centre for Applied Cryptologic Research, Mar. 1998. Preliminary version of [LMQ+03] at `http://www.cacr.math.uwaterloo.ca/techreports/1998/corr98-05.pdf`.

[LMQ+03] ———. *An efficient protocol for authenticated key agreement.* Designs, Codes and Cryptography, **28**:119–134, 2003.

[LN87] R. Lidl and H. Niederreiter. *Finite Fields*. Cambridge University Press, 1987.

[LZ94] G.-J. Lay and H. G. Zimmer. *Constructing elliptic curves with given group order over large finite fields*. Algorithmic Number Theory, pp. 250–263, 1994.

[McE87] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1987.

[Men93] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[Men07] ———. *Another look at HMQV*. Journal of Mathematical Cryptology, **1**:47–64, 2007. `http://eprint.iacr.org/2005/205`.

[Mil85] V. S. Miller. *Uses of elliptic curves in cryptography*. In H. C. Williams (ed.), *Advances in Cryptology — CRYPTO '85*, Lecture Notes in Computer Science 218, pp. 417–426. International Association for Cryptologic Research, Springer, Santa Barbara, California, Aug. 1985.

[MvOV97] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. `http://www.cacr.math.uwaterloo.ca/hac/`.

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

[MQV95] A. J. Menezes, M. Qu and S. A. Vanstone. *Some new key agreement protocols providing implicit authentication*. In *Selected Areas in Cryptography - SAC '95*, pp. 22–32. May 1995.

[MSV04] A. Muzereau, N. P. Smart and F. Vercauteren. *The equivalence between the DHP and DLP for elliptic curves used in practical applications*. LMS J. Comput. Math., **7**:50–72, Mar. 2004. `http://www.lms.ac.uk`.

[MT06] A. J. Menezes and E. Teske. *Cryptographic implications of Hess' generalized GHS attack*. Applicable Algebra in Engineering, Communication and Computing, **16**:439–460, 2006. `http://eprint.iacr.org/2004/235`.

[MW96] U. M. Maurer and S. Wolf. *Diffie-Hellman oracles*. In Koblitz [Kob96], pp. 268–282.

[Nac01] D. Naccache (ed.). *Topics in Cryptology — CT-RSA 2001*, Lecture Notes in Computer Science 2020. Springer, Apr. 2001.

[NNTW05] D. Naccache, P. Q. Nguyen, M. Tunstall and C. Whelan. *Experimenting with faults, lattices and the DSA*. In S. Vaudenay (ed.), *Public Key Cryptography — PKC 2005*, Lecture Notes in Computer Science 3386, pp. 16–28. International Association for Cryptologic Research, Springer, Jan. 2005.

[NR93] K. Nyberg and R. Rueppel. *A new signature scheme based on DSA giving message recovery.* In *1st ACM Conference on Computer and Communications Security*, pp. 58–61. ACM Press, 1993.

[NR96] ———. *Message recovery for signature schemes based on the discrete logarithm problem.* Designs, Codes and Cryptography, **7**:61–81, 1996.

[NS03] P. Q. Nguyen and I. E. Shparlinksi. *The insecurity of the elliptic curve digital signature algorithm with partially known nonces.* Designs, Codes and Cryptography, **30**:201–217, 2003. `http://eprint.iacr.org/2004/277`.

[Odl95] A. Odlyzko. *The future of integer factorization.* CryptoBytes, **1**(2):5–12, 1995.

[vOW94] P. C. van Oorschot and M. J. Wiener. *Parallel collision search with applications to hash functions and discrete logarithms.* In *2nd ACM Conference on Computer and Communications Security*, pp. 210–218. ACM Press, 1994.

[Pel06] J. Pelzl. *Exact cost estimates for ecc attacks with special-purpose hardware.* In *The 10th Workshop on Elliptic Curve Cryptography (ECC 2006)*. Sep. 2006. `http://www.cacr.math.uwaterloo.ca/conferences/2006/ecc2006/pelzl.pdf`.

[Pol78] J. Pollard. *Monte Carlo methods for index computation mod p.* Mathematics of Computation, **32**:918–924, 1978.

[PH78] S. C. Pohlig and M. E. Hellman. *An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance.* IEEE Transactions on Information Theory, **24**:106–110, 1978.

[PS96] D. Pointcheval and J. Stern. *Security proofs for signatures.* In U. M. Maurer (ed.), *Advances in Cryptology — EUROCRYPT '96*, Lecture Notes in Computer Science 1070, pp. 387–398. International Association for Cryptologic Research, Springer, May 1996.

[PV00] L. Pintsov and S. A. Vanstone. *Postal revenue collection in the digital age.* In Y. Frankel (ed.), *Financial Cryptography*, Lecture Notes in Computer Science 1962, pp. 105–120. Springer, Feb. 2000.

[PV05] P. Paillier and D. Vergnaud. *Discrete-log-based signatures may not be equivalent to discrete log.* In B. Roy (ed.), *Advances in Cryptology — ASIACRYPT 2005*, Lecture Notes in Computer Science 3788, pp. 1–20. International Association for Cryptologic Research, Springer, Dec. 2005.

[Rog06] P. Rogaway. *Formalizing human ignorance: Collision-resistant hashing without the keys.* In P. Q. Nguyen (ed.), *VIETCRYPT 2006*, Lecture Notes in Computer Science 4341, pp. 211–228. Springer, Sep. 2006. `http://eprint.iacr.org/2006/281`.

[Sat00] T. Satoh. *The canonical lift of an ordinary elliptic curve over a finite field and its point counting.* Journal of Ramanujan Mathematical Society, **15**:247–270, 2000.

[Sch85] R. Schoof. *Elliptic curves over finite fields and the computation of square roots mod p.* Mathematics of Computation, **44**:483–494, 1985.

[Sch91] C. P. Schnorr. *Efficient signature generation by smart cards.* Journal of Cryptology, **4**:161–174, 1991.

[Sem98] I. A. Semaev. *Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p.* Mathematics of Computation, **67**:353–356, 1998.

[Sho01] V. Shoup. *A proposal for an ISO standard for public key encryption*, Dec. 2001. `http://shoup.net/iso`.

[Sil85] J. Silverman. *The Arithmetic of Elliptic Curves.* Springer, 1985.

[Sma99] N. P. Smart. *The discrete logarithm problem on elliptic curves of trace one.* Journal of Cryptology, **12**:193–196, 1999.

[Sma01] ———. *The exact security of ECIES in the generic group model.* In B. Honary (ed.), *Coding and Cryptography*, Lecture Notes in Computer Science 2260, pp. 73–84. Springer, Dec. 2001.

[Sol01] J. A. Solinas. *Some computational speedups and bandwidth improvements for curves over prime fields.* In *The 5th Workshop on Elliptic Curve Cryptography (ECC 2001).* Oct. 2001. `http://www.cacr.math.uwaterloo.ca/conferences/2001/ecc/solinas.ps`.

[Sti06] D. R. Stinson. *Some observations on the theory of cryptographic hash functions.* Designs, Codes and Cryptography, **38**:259–277, 2006. `http://eprint.iacr.org/2001/020`.

[SA98] T. Satoh and K. Araki. *Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves.* Commentarii Mathematici Universitatis Sancti Pauli, **47**:81–92, 1998.

[SPMLS02] J. Stern, D. Pointcheval, J. Malone-Lee and N. P. Smart. *Flaws in applying proof methodologies to signature schemes.* In M. Yung (ed.), *Advances in Cryptology — CRYPTO 2002*, Lecture Notes in Computer Science 2442, pp. 93–110. International Association for Cryptologic Research, Springer, Aug. 2002.

[Van92] S. A. Vanstone. *Responses to NIST's proposal.* Communications of the ACM, **35**:50–52, 1992.

[Vau96] S. Vaudenay. *Hidden collisions on DSS.* In Koblitz [Kob96], pp. 83–88.

[Ver05] F. Vercauteren. *Advances in point counting.* In Blake et al. [BSS05], pp. 103–132. Chapter VI.

[WYY05a] X. Wang, A. C. Yao and F. Yao. *Cryptanalysis of SHA-1 hash function.* In National Institute of Standards and Technology (ed.), *1st Cryptographic Hash Workshop.* Oct. 2005. `http://www.csrc.nist.gov/pki/HashWorkshop/2005/program.htm`.

[WYY05b] X. Wang, Y. L. Yin and H. Yu. *Finding collisions in the full SHA-1.* In V. Shoup (ed.), *Advances in Cryptology — CRYPTO 2005*, Lecture Notes in Computer Science 3621, pp. 17–36. International Association for Cryptologic Research, Springer, Aug. 2005.

[WZ99] M. J. Wiener and R. J. Zuccherato. *Fast attacks on elliptic curve cryptosystems.* In S. Tavares and H. Meijer (eds.), *Selected Areas in Cryptography: SAC '98*, Lecture Notes in Computer Science 1556, pp. 190–200. Springer, 1999.